US007493248B2

US 7,493,248 B2

(12) **United States Patent**
Bobok et al.

(10) **Patent No.:** **US 7,493,248 B2**
(45) **Date of Patent:** **Feb. 17, 2009**

(54) **METHOD, SYSTEM AND PROGRAM PRODUCT SUPPORTING PHASE EVENTS IN A SIMULATION MODEL OF A DIGITAL SYSTEM**

(75) Inventors: **Gabor Bobok**, Niskayuna, NY (US);
Wolfgang Roesner, Austin, TX (US);
Derek E. Williams, Austin, TX (US)

(73) Assignee: **International Business Machines Corporation**, Armonk, NY (US)

(*) Notice: Subject to any disclaimer, the term of this patent is extended or adjusted under 35 U.S.C. 154(b) by 457 days.

(21) Appl. No.: **11/382,088**

(22) Filed: **May 8, 2006**

(65) **Prior Publication Data**

US 2007/0260441 A1 Nov. 8, 2007

(51) **Int. Cl.**
G06F 9/45 (2006.01)
(52) **U.S. Cl.** **703/22**
(58) **Field of Classification Search** 703/14, 703/22
See application file for complete search history.

(56) **References Cited**

U.S. PATENT DOCUMENTS

5,212,650 A	5/1993	Hooper et al.	
5,369,763 A	11/1994	Biles	
5,604,895 A	2/1997	Raimi	
6,117,181 A	9/2000	Dearth et al.	
6,195,627 B1	2/2001	Bargh et al.	
6,195,629 B1	2/2001	Bargh et al.	
6,202,042 B1	3/2001	Bargh et al.	
6,285,914 B1	9/2001	Bae et al.	
6,345,242 B1	2/2002	Dearth et al.	
6,393,426 B1	5/2002	Odom et al.	

6,519,612 B1	2/2003	Howard et al.	
6,546,532 B1	4/2003	Kerzman et al.	
6,581,191 B1	6/2003	Schubert et al.	
6,606,588 B1 *	8/2003	Schaumont et al.	.... 703/15
6,618,839 B1	9/2003	Beardslee et al.	
6,668,364 B2	12/2003	McElvain et al.	
6,687,710 B1	2/2004	Dey	
6,720,565 B2	4/2004	Innes et al.	
6,751,583 B1	6/2004	Clarke et al.	
6,754,763 B2	6/2004	Lin	
7,006,960 B2 *	2/2006	Schaumont et al.	.... 703/15
2002/0019969 A1	2/2002	Hellestrand et al.	
2002/0032559 A1	3/2002	Hellestrand et al.	

(Continued)

OTHER PUBLICATIONS

French et al., R.S. A General Method for Compiling Event-Driven Simulations, Proceedings of the 32nd ACM/IEEE Conference on Design Automation, Jan. 1995, pp. 1-6.*

Primary Examiner—Russell Frejd
(74) *Attorney, Agent, or Firm*—Dillon & Yudell LLP
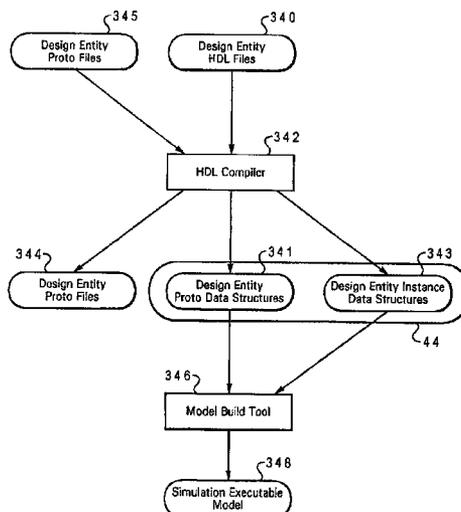
(57) **ABSTRACT**

According to a method of simulation processing, an instrumented simulation executable model of a design is built by compiling one or more hardware description language (HDL) files specifying one or more design entities within the design and one or more instrumentation entities and instantiating instances of the one or more instrumentation entities within instances of the one or more design entities. Operation of the design is then simulated utilizing the instrumented simulation executable model. Simulating operation includes each of multiple instantiations of the one or more instrumentation entities generating a respective external phase signal representing an occurrence of a particular phase of operation and instrumentation combining logic generating from external phase signals of the multiple instantiations of the one or more instrumentation entities an aggregate phase signal representing an occurrence of the particular phase.

3 Claims, 27 Drawing Sheets

U.S. PATENT DOCUMENTS

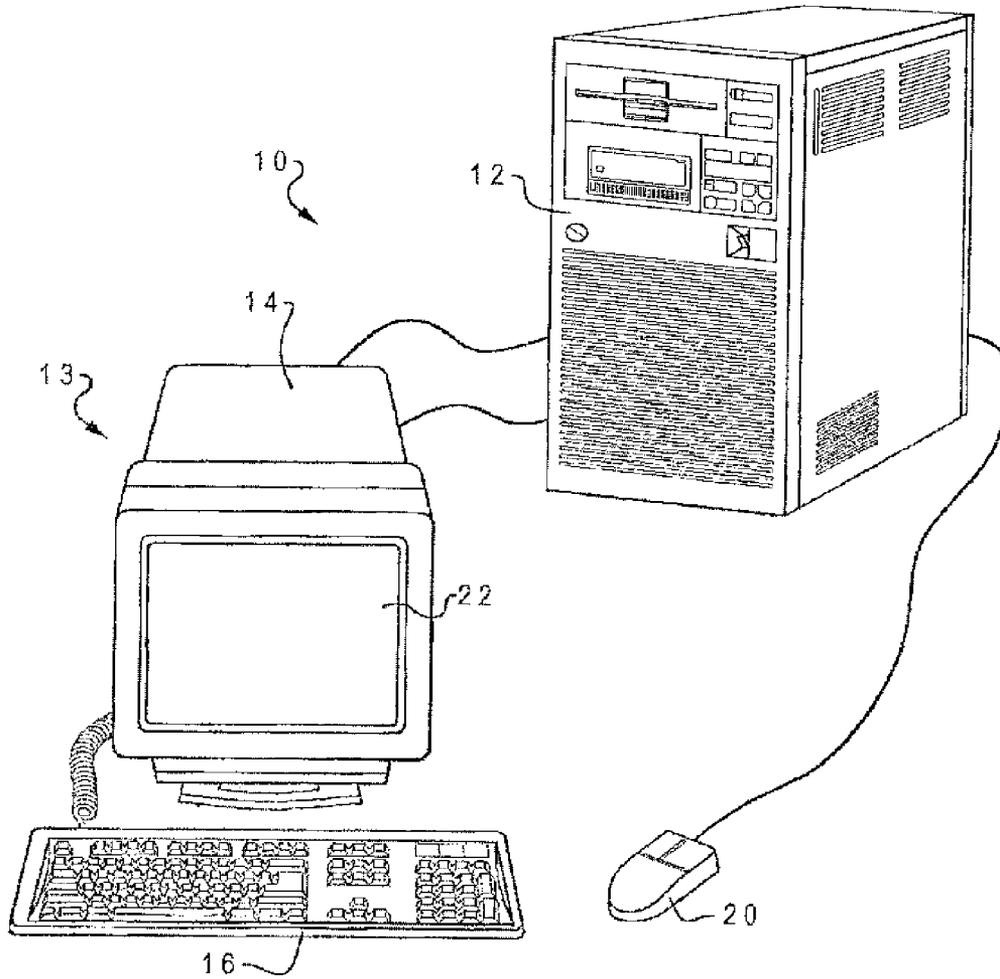|		
2002/0049576 A1	4/2002	Meyer
2002/0123874 A1	9/2002	Roesner et al.
2002/0128809 A1	9/2002	Roesner et al.
2002/0129333 A1	9/2002	Chandhoke et al.
2002/0152060 A1	10/2002	Tseng
2003/0005416 A1	1/2003	Henftling et al.
2003/0035375 A1	2/2003	Freeman
2003/0037222 A1	2/2003	Emberson et al.
2003/0101039 A1	5/2003	Gabele et al.
2003/0121011 A1	6/2003	Carter et al.
2003/0144828 A1	7/2003	Lin
2003/0188299 A1	10/2003	Broughton et al.
2003/0192032 A1	10/2003	Andrade et al.
2005/0102125 A1	5/2005	Tseng
2005/0125754 A1	6/2005	Schubert et al.
2005/0137840 A1	6/2005	Peck et al.

* cited by examiner

10

12

14

13

22

16

20

Fig. 1

Prior Art

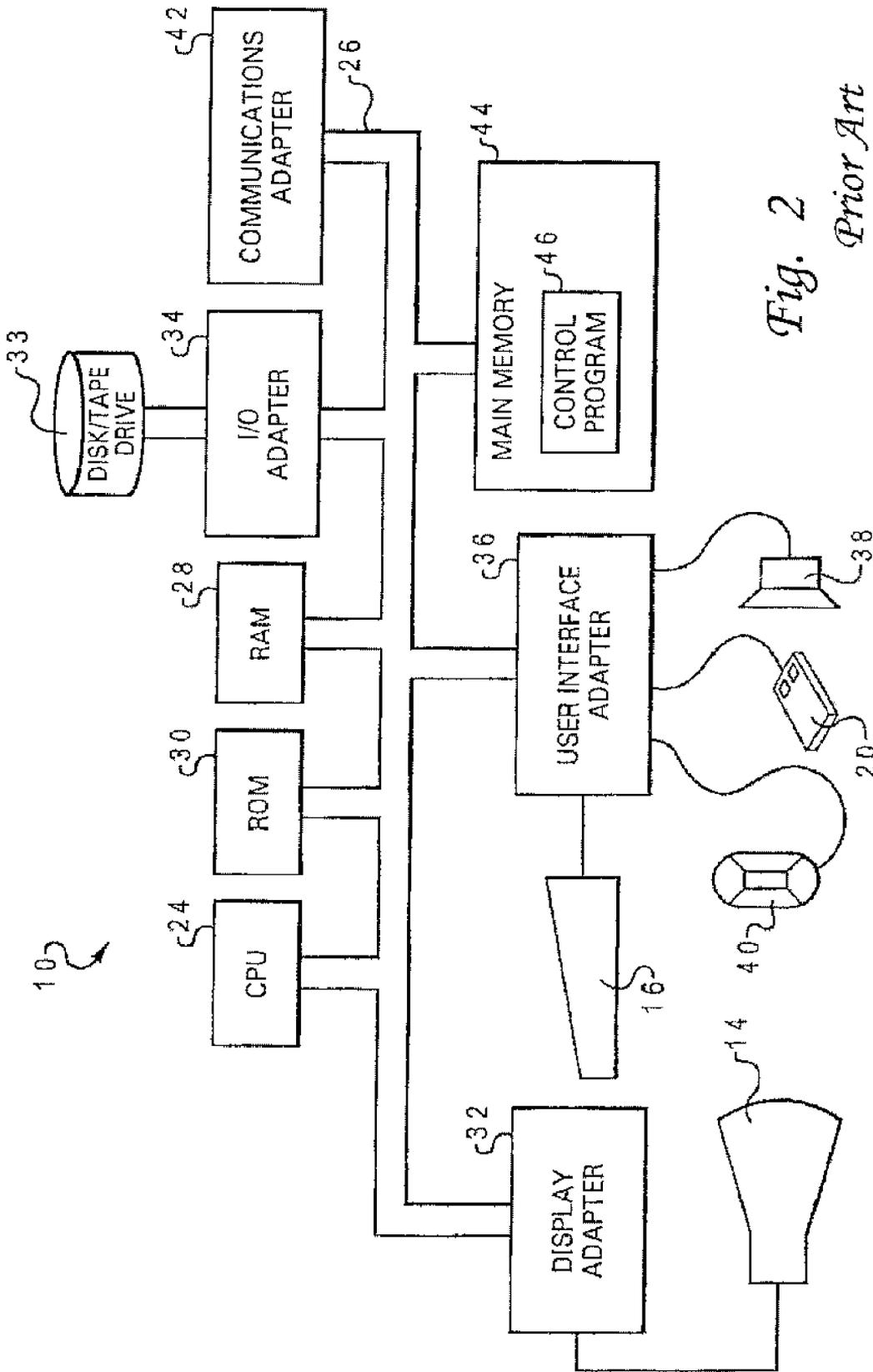Fig. 2 Prior Art

Fig. 3A

Fig. 3B

Fig. 3C

Fig. 3D

Fig. 4A

Fig. 4B

```
ENTITY FXUCHK IS                                                          ⎤ ⎤
                                                                          ⎥ ⎥
        PORT(   S_IN        :      IN std_ulogic;                         ⎥ ⎥
                Q_IN        :      IN std_ulogic;                         ⎥ ⎥
                R_IN        :      IN std_ulogic;                         ⎥ ⎥ 450
                clock       :      IN std_ulogic;                         ⎬ ⎥
                fails       :      OUT std_ulogic_vector(0 to 1);         ⎥ ⎥
                counts      :      OUT std_ulogic_vector(0 to 2);         ⎥ ⎥
                harvests    :      OUT std_ulogic_vector(0 to 1);         ⎥ ⎥
                phases      :      OUT std_ulogic_vector (0 to 0);        ⎦ ⎥
              );                                                            ⎥
```
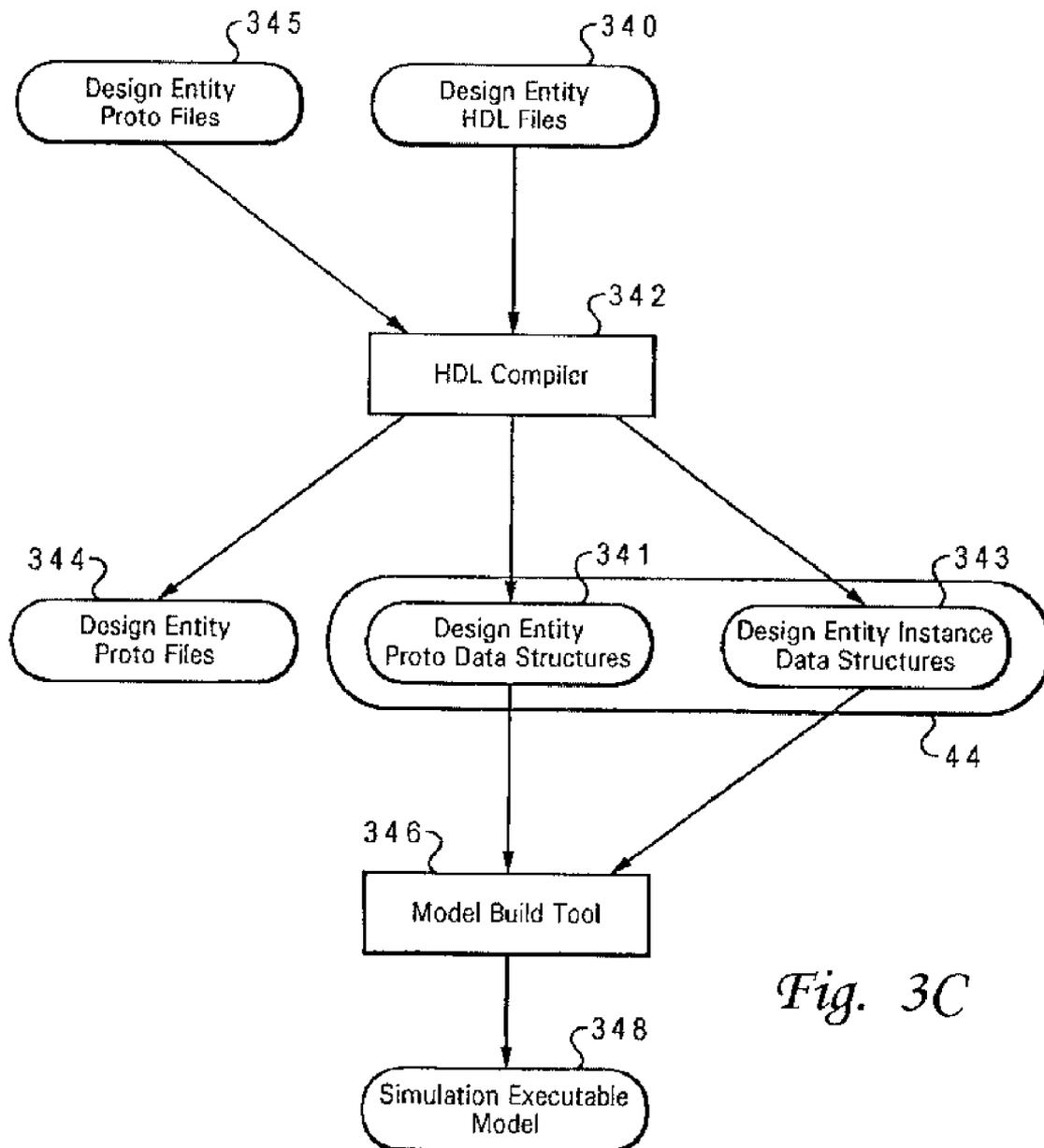
         ⎧ --!! BEGIN   ⎤ ⎥
   452 ⎨   ⎥ ⎥
         ⎩ --!! Design Entity: FXU;   ⎥ ⎥
                                                                          ⎥ ⎥
         ⎧ --!! Inputs   ⎥ ⎥
         ⎪ --!! S_IN   => B.C.S;   ⎥ ⎥
         ⎪ --!! Q_IN   => A.Q;   ⎥ ⎥
   453 ⎨ --!! R_IN   => R;   ⎥ ⎥
         ⎪ --!! CLOCK   => clock;   ⎥ ⎥
         ⎩ --!! End Inputs   ⎥ ⎥
                                                                          ⎥ ⎥
         ⎧ --!! Fail Outputs;   ⎥ ⎥ 440
         ⎪ --!! 0 : "Fail message for failure event 0";   ⎥ ⎬
   454 ⎨ --!! 1 : "Fail message for failure event 1";   ⎬ 451 ⎥
         ⎩ --!! End Fail Outputs;   ⎥ ⎥
                                                                          ⎥ ⎥
         ⎧ --!! Count Outputs;   ⎥ ⎥
         ⎪ --!! 0 : <event0> clock;   ⎥ ⎥
   455 ⎨ --!! 1 : <event1> clock;   ⎥ ⎥
         ⎪ --!! 2 : <event2> clock;   ⎥ ⎥
         ⎩ --!! End Count Outputs;   ⎥ ⎥
                                                                          ⎥ ⎥
         ⎧ --!! Harvest Outputs;   ⎥ ⎥
         ⎪ --!! 0 : "Message for harvest event 0";   ⎥ ⎥
   456 ⎨ --!! 1 : "Message for harvest event 1";   ⎥ ⎥
         ⎩ --!! End Harvest Outputs;   ⎥ ⎥
                                                                          ⎥ ⎥
         ⎧ --!! Phase Outputs;   ⎥ ⎥
   490 ⎨ --!! 0:instance_name, phasename;   ⎥ ⎥
         ⎩ --!! EndPhase Outputs;   ⎥ ⎥
                                                                          ⎥ ⎥
   457 ⎰ --!! End   ⎦ ⎥
                                                                             ⎥
         ARCHITECTURE example of FXUCHK IS   ⎥
                                                                             ⎥
         BEGIN   ⎤   ⎥
                                                              ⎥   ⎥
              ... HDL code for entity ...   ⎬ 458   ⎦
                                                              ⎥
         END;   ⎦   Fig. 4C

492
Phase File

345
Design Entity
Proto Files

460
Instrumentation
Entity Proto Files

461
Instrumentation
Entity HDL Files

340
Design Entity
HDL Files

462
HDL Compiler

468
Instrumentation
Entity Proto Files

344
Design Entity
Proto Files

463
Design Entity
Proto Data
Structures

465
Design Entity
Instance Data
Structures

466
Instrumentation
Entity Proto Data
Structures

464
Instrumentation Load Tool

463a
Instrumented
Design Entity Proto
Data Structures

465a
Instrumented Design
Entity Instance
Data Structures

467
Instrumentation
Entity Instance
Data Structures

420
Instrumentation
Logic Block

Model Build Tool

446

480
Instrumented
Simulation
Executable Model

Fig. 4D

Fig. 4E

Fig. 5A

Fig. 5B

Fig. 6

Prior Art

entity FSM : FSM

850

800

fsm_state(0 to 2)

801

802   fsm_clock

Fig. 7

Prior Art

```
ENTITY FSM IS

    PORT(
               ....ports for entity fsm....
          );

    ARCHITECTURE FSM OF FSM IS

    BEGIN

               ... HDL code for FSM and rest of the entity ...

               fsm_state(0 to 2) < = ... Signal 801 ...

    853 {  --!! Embedded FSM   : examplefsm;
    859 {  --!! clock           : (fsm_clock);
    854 {  --!! state_vector     : (fsm_state(0 to 2));
    855 {  --!! states           : (S0, S1, S2, S3, S4);
    856 {  --!! state_encoding    : ('000', '001', '010', '011', '100');
              --!! arcs          : (S0 = > S0, S0 = > S1, S0 = > S2,
    857 {     --!!                 (S1 = > S2, S1 = > S3, S2 = > S2,
              --!!                 (S2 = > S3, S3 = > S4, S4 = > S0);
    858 {  --!! End FSM;

                          ,
                          ,
                          ,
                          ,
                          ,
                          ,
                          ,

    END;
```

852 860

Fig. 8

entity FSM : FSM

850

800

fsm_state(0..2)

801

fsm_clock

802

900   903

fails(0..1)

counts(0..1)

903

$examplefsm : $examplefsm

902

Fig. 9

Fig. 10A

<instantiation identifier>.<instrumentation entity name>.<design entity name>.<eventname>

1030   1032   1034   1036

Fig. 10B

1030	1032	1034	1036	
X1	B3	X	COUNT1	1040
X1.Z	B1	Z	COUNT1	1041
X1.Z	B2	Z	COUNT1	1042
X2	B3	X	COUNT1	1043
X2.Z	B1	Z	COUNT1	1044
X2.Z	B2	Z	COUNT1	1045
Y	B4	Y	COUNT1	1046
Y.Z	B1	Z	COUNT1	1047
Y.Z	B2	Z	COUNT1	1048

Fig. 10C

<instantiation identifier>.<design entity name>.<eventname>

1030   1034   1036

Fig. 10D

*Fig. 11*A

```
                          1163         1165
--!! Inputs
--!! event_1108_in < = C.[B2.count.event_1108];       ~ 1161
--!! event_1124_in < = A.B.[B1.count.event_1124];     ~ 1162
--!! End Inputs
                          1164         1166
```

Fig. 11B

```
--!! Inputs
--!! event_1108_in < = C.[count.event_1108];       ~ 1171
--!! event_1124_in < = B.[count.event_1124];       ~ 1172
--!! End Inputs
```

Fig. 11C

Fig. 12A

```
ENTITY X IS

    PORT(    :
                 :
                 :
         );

ARCHITECTURE example of X IS

BEGIN
         .
         .
         .
         .
         .
    ... HDL code for X ...
         .
         .
         .
         .
```

1221 {
```
    Y:Y
    PORT MAP(    :
                      :
                  );
```

1222 {
```
    A < = ....
    B < = ....
    C < = ....
```

1223 {
```
    --!! [count, countname0, clock] < = Y.P;          1230   1232
    --!! Q  < = Y. [B1.count.count1] AND A;          1234
    --!! [fail, failname0, "fail msg"] < = Q XOR B;
    --!! [harvest, harvestname0, "harvest msg"] < = B AND C;
    --!! [phase, phase0, end_of_testcase] <= Y.R
ENDAND D;                                    1238          1236
```

1220

Fig. 12B

Fig. 13A

Fig. 13B

```
ENTITY DET IS

        PORT(   A          :      IN std_ulogic;
                B          :      IN std_ulogic_vector(0 to 5);
                C          :      IN std_ulogic;
                D          :      IN std_ulogic;
                                    .
                                    .
                                    .
                event_x    :      OUT std_ulogic_vector(0 to 2);
                x_here     :      OUT std_ulogic;
            );

    --!! BEGIN
    --!! Design Entity: LM;

    --!! Inputs
    --!! A      =>     A;
    --!! B      =>     P.Q.B;
    --!! C      =>     P.C;
    --!! D      =>     D;
    --!! End Inputs

    --!! Detections
    --!! <event_x>:event_x(0 to 2) [x_here];
    --!! End Detections

    --!! End;

    ARCHITECTURE example of DET IS

    BEGIN

        ... HDL code ...

    END;
```

Fig. 13c

1660

	1661	1662		
1:	X1	B3	X	COUNT1
2:	X1.Z	B1	Z	COUNT1
3:	X1.Z	B2	Z	COUNT1
4:	X2	B3	X	COUNT1
5:	X2.Z	B1	Z	COUNT1
6:	X2.Z	B2	Z	COUNT1
7:	Y	B4	Y	COUNT1
8:	Y.Z	B1	Z	COUNT1
9:	Y.Z	B2	Z	COUNT1

1663

Fig. 14

METHOD, SYSTEM AND PROGRAM PRODUCT SUPPORTING PHASE EVENTS IN A SIMULATION MODEL OF A DIGITAL SYSTEM

CROSS-REFERENCE TO RELATED APPLICATION

The present application is also related to co-pending U.S. patent application Ser. No. 11/006,093, which is assigned to the assignee of the present invention and incorporated herein by reference in its entirety.

BACKGROUND OF THE INVENTION

1. Technical Field

The present invention relates in general to designing and simulating digital devices, modules and systems, and in particular, to a method and system for computer simulation of digital devices, modules and systems utilizing a hardware description language (HDL) model.

2. Description of the Related Art

Verifying the logical correctness of a digital design and debugging the design, if necessary, are very important steps in most digital design processes. Logic networks are tested either by actually building networks or by simulating networks on a computer. As logic networks become highly complex, it becomes necessary to simulate a design before the design is actually built. This is especially true when the design is implemented as an integrated circuit, since the fabrication of integrated circuits requires considerable time and correction of mistakes is quite costly. The goal of digital design simulation is the verification of the logical correctness of the design.

In a typical automated design process that is supported by a conventional electronic computer-aided design (ECAD) system, a designer enters a high-level description utilizing a hardware description language (HDL), such as VHDL, producing a representation of the various circuit blocks and their interconnections. The ECAD system compiles the design description into a format that is best suited for simulation. A simulator is then utilized to verify the logical correctness of the design prior to developing a circuit layout.

A simulator is typically a software tool that operates on a digital representation, or simulation model of a circuit, and a list of input stimuli (i.e., testcase) representing inputs of the digital system. A simulator generates a numerical representation of the response of the circuit, which may then either be viewed on the display screen as a list of values or further interpreted, often by a separate software program, and presented on the display screen in graphical form. The simulator may be run either on a general-purpose computer or on another piece of electronic apparatus, typically attached to a general purpose computer, specially designed for simulation. Simulators that run entirely in software on a general-purpose computer will hereinafter be referred to as "software simulators". Simulators that are run with the assistance of specially designed electronic apparatus will hereinafter be referred to as "hardware simulators".

Usually, software simulators perform a very large number of calculations and operate slowly from the user's point of view. In order to optimize performance, the format of the simulation model is designed for very efficient use by the simulator. Hardware simulators, by nature, require that the simulation model comprising the circuit description be communicated in a specially designed format. In either case, a

translation from an HDL description to a simulation format, hereinafter referred to as a simulation executable model, is required.

The present invention appreciates that certain events, such as reaching the end of a particular simulation testcase, are difficult to detect in a conventional simulation executable model. For example, in conventional simulation environments, a designer may detect the occurrence of the end of a testcase by incorporating into the simulation executable model special-purpose, hand-built logic to monitor specific signals for which a particular set of values indicates that the end of a testcase has been reached. In some cases, the complexity of such monitoring logic may approach the complexity of the simulated circuit being monitored. Consequently, the present invention appreciates that it would be useful and desirable to provide a method, system and program product that improve the detection and visibility of certain events, such as end of testcase events, within simulation executable models.

SUMMARY OF THE INVENTION

According to a method of simulation processing, an instrumented simulation executable model of a design is built by compiling one or more hardware description language (HDL) files specifying one or more design entities within the design and one or more instrumentation entities and instantiating instances of the one or more instrumentation entities within instances of the one or more design entities. Operation of the design is then simulated utilizing the instrumented simulation executable model. Simulating operation includes each of multiple instantiations of the one or more instrumentation entities generating a respective external phase signal representing an occurrence of a particular phase of operation and instrumentation combining logic generating from external phase signals of the multiple instantiations of the one or more instrumentation entities an aggregate phase signal representing an occurrence of the particular phase.

All objects, features, and advantages of the present invention will become apparent in the following detailed written description.

BRIEF DESCRIPTION OF THE DRAWINGS

The novel features believed characteristic of the invention are set forth in the appended claims. The invention itself however, as well as a preferred mode of use, further objects and advantages thereof, will best be understood by reference to the following detailed description of an illustrative embodiment when read in conjunction with the accompanying drawings, wherein:

FIG. 1 is a pictorial representation of a data processing system;

FIG. 2 depicts a representative hardware environment of the data processing system illustrated in FIG. 1;

FIG. 3A is a simplified block diagram illustrating a digital design entity that may be instrumented in accordance with the teachings of the present invention;

FIG. 3B is a diagrammatic representation depicting a simulation model that may be instrumented in accordance with the teachings of the present invention;

FIG. 3C is a flow diagram illustrating of a model build process that may be implemented in accordance with the teachings of the present invention;

FIG. 3D is a block diagram depicting data structures that may be instrumented in accordance with the teachings of the present invention;

3

FIG. **4A** is a simplified block diagram representative of an instrumentation entity;

FIG. **4B** is a simplified block diagram of a simulation model instrumented in accordance with the teachings of the present invention;

FIG. **4C** illustrates exemplary sections of HDL syntax that maybe utilized in accordance with the teachings of the present invention;

FIG. **4D** is a flow diagram depicting a model build process in accordance with the teachings of the present invention;

FIG. **4E** is a block diagram representation of memory data structures constructed in accordance with the teachings of the present invention;

FIG. **5A** is a logic diagram representation of a runtime disable mechanism in accordance with the teachings of the present invention;

FIG. **5B** is a block diagram representation of functional units utilized to execute the method and system of the present invention on a hardware simulator in accordance with the teachings of the present invention;

FIG. **6** illustrates a conventional finite state machine that may be instrumented with an embedded checker in accordance with the teachings of the present invention;

FIG. **7** depicts a conventional finite state machine design entity;

FIG. **8** illustrates a hardware description language file including embedded instrumentation in accordance with the teachings of the present invention;

FIG. **9** depicts a hardware description language design entity included embedded instrumentation in accordance with the teachings of the present invention;

FIG. **10A** is a block diagram illustrating a simulation model containing a number of design and instrumentation entities;

FIG. **10B** depicts a data structure for declaring an event within a simulation model in accordance with one embodiment of the present invention;

FIG. **10C** illustrates a list of extended event data structures for the simulation model in FIG. **10A**;

FIG. **10D** depicts a data structure for declaring an event within a simulation model in accordance with an alternate embodiment of the present invention;

FIG. **11A** is a block diagram illustrating a simulation model in which the hierarchical event processing of the present invention is applicable;

FIG. **11B** depicts a set of input port mapping comments for performing hierarchical processing of simulation model events in accordance with a first embodiment of the present invention;

FIG. **11C** illustrates a set of input port mapping comments for performing hierarchical processing of simulation model events in accordance with a second embodiment of the present invention;

FIG. **12A** depicts a representative target design entity with an instrumentation entity containing random instrumentation logic implemented in accordance with the teachings of the present invention;

FIG. **12B** illustrates an exemplary HDL file for implementing instrumentation logic within an HDL design entity in accordance with the teachings of the present invention;

FIG. **13A** depicts an exemplary design entity containing a multi-bit simulation signal;

FIG. **13B** illustrates a design entity wherein signal injection is implemented in accordance with the teachings of the present invention;

4

FIG. **13C** depicts an exemplary HDL source file that describes an instrumentation entity in accordance wit the teachings of the present invention; and

FIG. **14** is a block diagram depicting an eventlist file for count events during a simulation run of a simulation model.

DETAILED DESCRIPTION OF A PREFERRED EMBODIMENT

The present invention provides for accurate and comprehensive monitoring of a digital circuit design in which a designer creates instrumentation modules utilizing the same hardware description language (HDL) as utilized for the design itself. HDLs, while suited to the needs of digital designers can also be effectively utilized for a number of checking functions. In accordance with the present invention, instrumentation modules are utilized to monitor specified design parameters while not becoming compiled as an integral part of the design itself. Furthermore, since the instrumentation modules are written in the same HDL as utilized in the actual design, such modules are platform and simulator independent. Unlike checking done with C or C++ programs, HDL instrumentation can be compiled and run directly without loss of performance on hardware simulators.

With reference now to the figures, and in particular with reference to FIG. **1**, there is depicted a pictorial representation of a data processing system **10** with which the present invention may be advantageously utilized. As illustrated, data processing system **10** comprises a workstation **12** to which one or more nodes **13** are connected. Workstation **12** preferably comprises a high performance multiprocessor computer, such as one of the POWER line of computer systems available from International Business Machines (IBM) Corporation of Armonk, N.Y. Workstation **12** preferably includes nonvolatile and volatile internal storage for storing software applications comprising an ECAD system, which can be utilized to develop and verify a digital circuit design in accordance with the method and system of the present invention. As depicted, nodes **13** include a display device **14**, a keyboard **16**, and a mouse **20**. The ECAD software applications executed within workstation **12** preferably display a graphic user interface (GUI) within display screen **22** of display device **14** with which a digital circuit designer can interact using a keyboard **16** and mouse **20**. Thus, by entering appropriate inputs utilizing keyboard **16** and mouse **20**, the digital circuit designer is able to develop and verify a digital circuit design according to the method described further hereinbelow.

FIG. **2** is a more detailed block diagram of data processing system **10**. As illustrated, data processing system **10** includes one or more Central Processing Units (CPUs) **24**, such as a conventional microprocessor, and a number of other components interconnected via system interconnect **26**. Although not depicted in FIG. **2**, CPUs such as CPU **24** typically include a control unit that organizes data and program storage in a computer memory and transfers the data and other information between the various parts of the computer system. CPUs also generally include one or more arithmetic logic units that execute arithmetical and logical operations, such as addition, comparison, multiplication and so forth.

Data processing system **10** further includes a random-access memory (RAM) **28**, a read-only memory (ROM) **30**, a display adapter **32** supporting connection of a display device **14**, and an I/O adapter **34** for connecting peripheral devices (e.g., disk and tape drives **33**). Data processing system **10** further includes a communications adapter **42** for connecting data processing system **10** to a communications network and a user interface adapter **36** for connecting keyboard **16**, mouse

20, speaker **38**, microphone **40**, and/or other user interface devices to system interconnect **26**.

As will be appreciated by those skilled in the art, data processing system **10** operates under the control of an operating system (e.g., AIX) and one or more other programs, which may reside in any suitable computer-readable media such as RAM **28**, ROM **30**, a magnetic disk, magnetic tape, or optical disk (the last three being located in disk and tape drives **33**). These programs include a control program **46** that resides within main memory **44** and contains program code that, when processed by CPU **24**, carries out the operations depicted in FIG. **4**D and FIG. **4**E described herein.

Simulated digital circuit design models are comprised of at least one and usually many sub-units referred to hereinafter as design entities. FIG. **3**A is a block diagram representation of an exemplary design entity **300** in which the method and system of the present invention may be implemented. Design entity **300** is defined by a number of components: an entity name, entity ports, and a representation of the function performed by design entity **300**. Each entity within a given model has a unique name (not explicitly shown in FIG. **3**A) that is declared in the HDL description of each entity. Furthermore, each entity typically contains a number of signal interconnections, known as ports, to signals outside the entity. These outside signals may be primary input/outputs (I/Os) of an overall design or signals connecting to other entities within an overall design.

Typically, ports are categorized as belonging to one of three distinct types: input ports, output ports, and bi-directional ports. Design entity **300** is depicted in as having a number of input ports **303** that convey signals into design entity **300**. Input ports **303** are connected to input signals **301**. In addition, design entity **300** includes a number of output ports **306** that convey signals out of design entity **300**. Output ports **306** are connected to a set of output signals **304**. Bi-directional ports **305** are utilized to convey signals into and out of design entity **300**. Bi-directional ports **305** are in turn connected to a set of bi-directional signals **309**. An entity, such as design entity **300**, need not contain ports of all three types, and in the degenerate case, contains no ports at all. To accomplish the connection of entity ports to external signals, a mapping technique, known as a "port map", is utilized. A port map (not explicitly depicted in FIG. **3**A) consists of a specified correspondence between entity port names and external signals to which the entity is connected. When building a simulation model, ECAD software is utilized to connect external signals to appropriate ports of the entity according to a port map specification.

Finally, design entity **300** contains a body section **308** that describes one or more functions performed by design entity **300**. In the case of a digital design, body section **308** contains an interconnection of logic gates, storage elements, etc., in addition to instantiations of other entities. By instantiating an entity within another entity, a hierarchical description of an overall design is achieved. For example, a microprocessor may contain multiple instances of an identical functional unit. As such, the microprocessor itself will often be modeled as a single entity. Within the microprocessor entity, multiple instantiations of any duplicated functional entities will be present.

Referring now to FIG. **3**B, there is illustrated a diagrammatic representation of an exemplary simulation model **329** that may be utilized in a preferred embodiment of the present invention. Simulation model **329** includes multiple hierarchical design entities. For visual simplicity and clarity, the ports and signals interconnecting the entities within simulation model **329** have not been explicitly shown. In any model, one

and only one entity is the so-called "top-level entity". A top-level entity **320** is that entity which encompasses all other entities within simulation model **329**. That is to say, top-level entity **320** instantiates, either directly or indirectly, all descendant entities within a design. Simulation model **329** consists of top-level entity **320** which directly instantiates two instances, **321a** and **321b**, of an FXU entity **321** and a single instance of an FPU entity **322**. Each instantiation has an associated description, which contains an entity name and a unique instantiation name. For top-level entity **320**, description **310** is labeled "TOP:TOP". Description **310** includes an entity name **312**, labeled as the "TOP" preceding the colon, and also includes an instantiation name **314**, labeled as the "TOP" following the colon.

It is possible for a particular entity to be instantiated multiple times as is depicted with instantiations **321a** and **321b** of FXU entity **321**. Instantiations **321a** and **321b** are distinct instantiations of FXU entity **321** with instantiation names FXU0 and FXU1 respectively. Top-level entity **320** is at the highest level within the hierarchy of simulation model **329**. An entity that instantiates a descendant entity will be referred to hereinafter as an "ancestor" of the descendant entity. Top-level entity **320** is therefore the ancestor that directly instantiates FXU entity instantiations **321a** and **321b**. At any given level of a simulation model hierarchy, the instantiation names of all instantiations must be unique.

In addition to FXU entity instantiations **321a** and **321b**, top-level entity **320** directly instantiates a single instance of a FPU entity **322** having an entity name FPU and instantiation name FPU0. Within an entity description, it is common for the entity name to match the instantiation name when only one instance of that particular entity is placed at a given level of a simulation model hierarchy. However, this is not required as shown by entity **322** (instantiation name FPU0, entity name FPU).

Within instantiation **321a** of FXU entity **321**, single instance entities **325a** and **326a** of entity A **325** and entity B **326** respectively, are directly instantiated. Similarly, instantiation **321b** of the same FXU entity contains instantiations **325b** and **326b** of entity A **325** and entity B **326** respectively. In a similar manner, instantiation **326a** and instantiation **326b** each directly instantiate a single instance of entity C **327** as entities **327a** and **327b** respectively. The nesting of entities within other entities can continue to an arbitrary level of complexity provided that all entities instantiated, whether singly or multiply, have unique entity names and the instantiation names at any given level of the hierarchy are unique with respect to one another. Each entity is constructed from one or more HDL files that contain the information necessary to describe the entity.

Associated with each entity instantiation is a so called "instantiation identifier". The instantiation identifier for a given instantiation is a string consisting of the enclosing entity instantiation names proceeding from the top-level entity instantiation name. For example, the instantiation identifier of instantiation **327a** of entity C **327** within instantiation **321a** of FXU entity **321** is "TOP.FXU0.B.C". This identifier serves to uniquely identify each instantiation within a simulation model.

Referring now to FIG. **3**C, there is depicted a flow diagram of a model build process which may be implemented in a preferred embodiment of the present invention. The process begins with one or more design entity HDL source code files **340** and, potentially, one or more design entity intermediate format files **345**, hereinafter referred to as "proto files" **345**, available from a previous run of an HDL compiler **342**. HDL compiler **342** processes HDL file(s) **340** beginning with the

7

top level entity of a simulation model and proceeding in a recursive fashion through all HDL or proto file(s) describing a complete simulation model. For each of HDL files **340** during the compilation process, HDL compiler **342**, examines proto files **345** to determine if a previously compiled proto file is available and consistent. If such a file is available and consistent, HDL compiler **342** will not recompile that particular file, but will rather refer to an extant proto file. If no such proto file is available or the proto file is not consistent, HDL compiler **342** explicitly recompiles the HDL file **340** in question and creates a proto file **344**, for use in subsequent compilations. Such a process will be referred to hereinafter as "incremental compilation" and can greatly speed the process of creating a simulation executable model **348**. Incremental compilation is described in further detail hereinbelow. Once created by HDL compiler **342**, proto files **344** are available to serve as proto files **345** in subsequent compilations.

In addition to proto files **344**, HDL compiler **342** also creates two sets of data structures, design entity proto data structures **341** and design entity instance data structures **343**, in memory **44** of computer system **10**. Design entity proto data structures **341** and design entity instance data structures **343**, serve as a memory image of the contents of a simulation executable model **348**. Data structures **341** and **343** are passed, via memory **44**, to a model build tool **346** that processes data structures **341** and **343** into simulation executable model **348**.

It will be assumed hereinafter that each entity is described by a single HDL file. Depending on convention or the particular HDL in which the current invention is practiced, this restriction may be required. However, in certain circumstances or for certain HDLs it is possible to describe an entity by utilizing more than one HDL file. Those skilled in the art will appreciate and understand the extensions necessary to practice the present invention if entities are permitted to be described by multiple HDL files. Furthermore, it will be assumed that there is a direct correspondence, for each entity, between the entity name and both of the following: the name of the HDL file representing the entity, and the name of the proto file for the entity.

In the following description, an HDL source code file corresponding to a given entity will be referred to by an entity name followed by ".vhdl". For example, the HDL source code file that describes top-level entity **320** will be referred to as TOP.vhdl. This labeling convention serves as a notational convenience only and should not be construed as limiting the applicability of the present invention to HDLs other than VHDL.

Returning to FIG. 3B, it can be seen that each entity may instantiate, either directly or indirectly, one or more other entities. For example, the FXU entity directly instantiates A entity **325** and B entity **326**. Furthermore, B entity **326** directly instantiates C entity **327**. Therefore, FXU entity **321** instantiates, directly or indirectly, A entity **325**, B entity **326** and C entity **327**. Those entities, that are directly or indirectly instantiated by another entity, will be referred to hereinafter as "descendants". The descendants of top level entity **320** are FXU entity **321**, FPU entity **322**, A entity **325**, B entity **326**, and C entity **327**. It can be seen that each entity has a unique set of descendants and that each time an entity is instantiated, a unique instance of the entity and its descendants is created. Within simulation model **329**, FXU entity **321** is instantiated twice, FXU:FXU0 **321a** and FXU:FXU1 **321b**, by top-level entity **320**. Each instantiation of FXU entity **321** creates a unique set of instances of the FXU, A, B, and C entities.

For each entity, it is possible to define what is referred to as a "bill-of-materials" or BOM. A BOM is a list of HDL files

8

having date and time stamps of the entity itself and the entity's descendants. Referring again to FIG. 3C, the BOM for an entity is stored in proto file **344** after compilation of the entity. Therefore, when HDL compiler **342** compiles a particular HDL source code file among HDL files **340**, a proto file **344** is generated that includes a BOM listing the HDL files **340** that constitute the entity and the entity's descendants, if any. The BOM also contains the date and time stamp for each of the HDL files referenced as each appeared on disk/tape **33** of computer system **10** when the HDL file was being compiled.

If any of the HDL files constituting an entity or the entity's descendants is subsequently changed, proto file **344** will be flagged as inconsistent and HDL compiler **342** will recompile HDL file **340** on a subsequent re-compilation as will be described in further detail below. For example, going back to FIG. 3B, the HDL files referenced by the BOM of FXU entity **321** are FXU.vhdl, A.vhdl, B.vhdl and C.vhdl, each with appropriate date and time stamps. The files referenced by the BOM of top-level entity **320** are TOP.vhdl, FXU.vhdl, A.vhdl, B.vhdl, C.vhdl, and FPU.vhdl with appropriate date and time stamps.

Returning to FIG. 3C, HDL compiler **342** creates an image of the structure of a simulation model in main memory **44** of computer system **10**. This memory image is comprised of the following components: "proto" data structures **341** and "instance" data structures **343**. A proto is a data structure that, for each entity in the model, contains information about the ports of the entity, the body contents of the entity, and a list of references to other entities directly instantiated by the entity (in what follows, the term "proto" will be utilized to refer to the in-memory data structure described above and the term "proto file" will be utilized to describe intermediate format file(s) **344**). Proto files **344** are therefore on-disk representations of the in-memory proto data structure produced by HDL compiler **342**.

An instance data structure is a data structure that, for each instance of an entity within a model, contains the instance name for the instance, the name of the entity the instance refers to, and the port map information necessary to interconnect the entity with external signals. During compilation, each entity will have only one proto data structure, while, in the case of multiple instantiations of an entity, each entity may have one or more instance data structures.

In order to incrementally compile a model efficiently, HDL compiler **342** follows a recursive method of compilation in which successive entities of the model are considered and loaded from proto files **345** if such files are available and are consistent with the HDL source files constituting those entities and their descendants. For each entity that cannot be loaded from existing proto files **345**, HDL compiler **342** recursively examines the descendants of the entity, loads those descendant entities available from proto file(s) **345** and creates, as needed, proto files **344** for those descendants that are inconsistent with proto files **345**. Pseudocode for the main control loop of HDL compiler **342** is shown below (the line numbers to the right of the pseudocode are not a part of the pseudocode, but merely serve as a notational convenience).

```
process_HDL_file(file)                                      5
{                                                          10
    if (NOT proto_loaded(file)) {                          15
        if (exists_proto_file(file) AND check_bom(file)) { 20
            load_proto(file);                              25
        } else {                                           30
            parse_HDL_file(file)                           35
```

-continued

```
        for (all instances in file) {              40
                process_HDL_file(instance);          45
            }                                      50
            create_proto(file);                    55
            write_proto_file(file);                60
        }                                          65
    }                                              70
    create_instance(file);                         75
}                                                  80
```

When compiler **342** is initially invoked, no proto data structures **341** or instance data structures **343** are present in memory **44** of computer system **10**. The main control loop, routine process_HDL_file( ) (line 5), is invoked and passed the name of the top level entity by means of parameter "file". The algorithm first determines if a proto data structure for the current entity is present in memory **44** by means of routine proto_loaded( ) (line 15). Note that the proto data structure for the top level entity will never be present in memory because the process starts without any proto data structures loaded into memory **44**. If a matching proto data structure is present in memory **44**, instance data structures for the current entity and the current entity's descendants, if any, are created as necessary in memory **44** by routine create_instance( ) (line 75).

However, if a matching proto data structure is not present in memory **44**, control passes to line 20 where routine exists_proto_file( ) examines proto files **345** to determine if a proto file exists for the entity. If and only if a matching proto file exists, routine check_bom( ) is called to determine whether proto file **345** is consistent. In order to determine whether the proto file is consistent, the BOM for the proto file is examined. Routine check_bom( ) examines each HDL source code file listed in the BOM to determine if the date or time stamps for the HDL source code file have changed or if the HDL source code file has been deleted. If either condition occurs for any file in the BOM, the proto file is inconsistent and routine check_bom( ) fails. However, if check_bom( ) is successful, control is passed to line 25 where routine load_proto( ) loads the proto file and any descendant proto files into memory **44**, thus creating proto data structures **341** for the current entity and the current entity's descendants, if any. The construction of process_HDL_file( ) ensures that once a proto file has been verified as consistent, all of its descendant proto files, if any, are also consistent.

If the proto file is either non-existent or is not consistent, control passes to line 35 where routine parse_HDL_file( ) loads the HDL source code file for the current entity. Routine parse_HDL_file( ) (line 35) examines the HDL source code file for syntactic correctness and determines which descendant entities, if any, are instantiated by the current entity. Lines **40**, **45**, and **50** constitute a loop in which the routine process_HDL_file( ) is recursively called to process the descendent entities that are called by the current entity. This process repeats recursively traversing all the descendants of the current entity in a depth-first fashion creating proto data structures **341** and proto data files **344** of all descendants of the current entity. Once the descendant entities are processed, control passes to line 55 where a new proto data structure is created for the current entity in memory **44** by routine create_proto( ). Control then passes to line 60 where a new proto file **344**, including an associated BOM, is written to disk **33** by routine write_proto_file( ). Finally, control passes to line 75 where routine create_instance( ) creates instance data structures **343** for the current entity and any descendant entities as

necessary. In this manner, process_HDL_file( ) (line 5) recursively processes the entire simulation model creating an in-memory image of the model consisting of proto data structures **341** and instance data structures **343**.

With reference now to FIG. **3D** there is depicted a block diagram representing compiled data structures, which may be implemented in a preferred embodiment of the present invention. Memory **44** contains proto data structures **361**, one for each of the entities referred to in simulation model **329**. In addition, instantiations in simulation model **329** are represented by instance data structures **362**. Instance data structures **362** are connected by means of pointers indicating the hierarchical nature of the instantiations of the entities within simulation model **329**. Model build tool **346** in FIG. **3C** processes the contents of memory **44** into memory data structures in order to produce simulation executable model **348**.

In order to instrument simulation models, the present invention makes use of entities known as "instrumentation entities," which are in contrast to the entities constituting a design which are referred to herein as "design entities". As with design entities, instrumentation entities are described by one or more HDL source code files and consist of a number of signal ports, a body section, and an entity name. In what follows, it will be assumed that an instrumentation entity is described by a single HDL file. Those skilled in the art will appreciate and understand extensions necessary to practice the current invention for an instrumentation entity that is described by multiple HDL files. Each instrumentation entity is associated with a specific design entity referred to hereinafter as the "target entity".

With reference now to FIG. **4A**, there is illustrated a block diagram representation of an instrumentation entity **409**. Instrumentation entity **409** includes a number of input ports **400** that are connected to signals **401** within a target entity (not depicted in FIG. **4A**). A body section **402** contains logic necessary to detect occurrences of specified conditions within the target entity and generate simulation model "events" with respect to signals **401**. Four distinct types of events may be generated: "count" events, "fail" events, "harvest" events, and "phase" events, each described below in turn. Body section **402** contains internal logic for detecting occurrences of conditions precipitating generation of these events. A set of multi-bit output ports **403**, **404**, **405** and **430** are connected to external instrumentation logic (depicted in FIG. **4B**) by external signals **406**, **407**, **408** and **432**, respectively. Output ports **403**, **404**, **405** and **430** thus provide the connection from the internal logic in body section **402** to the external instrumentation logic which is utilized to indicate the occurrence of count, fail, harvest, and phase events.

A fail or failure event is a sequence of signal values that indicate a failure in the correct operation of the simulation model. Each instrumentation entity monitors the target entity for any desired number of failure events. Each occurrence of a failure event is assigned to a particular signal bit on output port **403**. Logic within body section **402** produces an active high pulse on a specified bit of signal **403** when a failure condition is detected. Such activation of signal **403** is defined as a failure event. This error indication is conveyed by external signal **406** to external instrumentation logic (depicted in FIG. **4B** as external instrumentation logic block **420**), which flags the occurrence of the failure event.

A count event is a sequence of signal values that indicate the occurrence of an event within a simulation model for which it would be advantageous to maintain a count. Count events are utilized to monitor the frequency of occurrence of specific sequences within a simulation model. Each instrumentation entity can monitor the target entity for any desired

number of count events. Each count event is assigned to a particular signal bit on output port **405**. Logic block **402** contains the logic necessary to detect the occurrence of the desired count events and produces an active high pulse on the specified bit of signal **405** when a count event is detected. This count indication is conveyed by external signal **408** to instrumentation logic, which contains counters utilized to record the number of occurrences of each count event.

The third event type, a harvest event, is a sequence of signal values that indicate the occurrence of a specific operative circumstance, which would be advantageous to be able to reproduce. When a harvest event occurs, a register within an external instrumentation logic block is loaded to indicate at what point within a simulation run the event occurred, and a flag is set to indicate the occurrence of the specific circumstance. The details of the simulation run can thus be saved in order to recreate the specific circumstance monitored by the harvest event. Logic block **402** contains the logic necessary to detect the harvest events.

Each instrumentation entity can detect any desired number of harvest events that are each assigned to a particular signal bit on output port **404**. Logic within block **402** produces an active high pulse on the specified bit of signal **404** when a harvest event is detected. This harvest event detection is conveyed by external signal **407** to external instrumentation logic that contains a register and flag for each harvest event. The register is utilized to record at which point in the simulation run the harvest event occurred, and the flag is utilized to indicate the occurrence.

A phase event is an indication that one or more signal states defining a specified phase of operation of the simulation model are present in a target design entity. Each instrumentation entity monitors the target design entity for any desired number of phase events. Each occurrence of a phase event is assigned to a particular signal bit on output port **430**. Logic within body section **402** produces an active high pulse on a specified bit of signal **430** when the signal state(s) signifying the phase event are detected within the target design entity. Such activation of signal **430** is defined as a phase event. This phase indication is conveyed by external signal **432** to external instrumentation logic (depicted in FIG. **4B** as external instrumentation logic block **420**), which flags the occurrence of the phase event.

With reference now to FIG. **4B**, there is depicted a block diagram representation of simulation model **329** instrumented in accordance with the teachings of the present invention. As can be seen in FIG. **4B**, instances **410** and **411** of an instrumentation entity FXUCHK are utilized to monitor instances **321a** and **321b** of an FXU entity. For each FXU instantiations of **321a** and **321b**, an FXUCHK instantiation, **410** and **411** respectively, is automatically generated by the mechanism of the present invention. In a similar fashion, instrumentation entity FPUCHK **412** is instantiated to monitor FPU entity **322**.

As depicted in FIG. **4B**, entity FXUCHK monitors signals Q **372**, R **376**, and S **374** within each of instances **321a** and **321b** of the FXU entity. Signal Q **372** is a signal within the instances **325a** and **325b** of descendant entity A. Likewise, signal S **374** is a signal within descendant entity C that resides within descendant entity B. Finally, signal R **376** occurs directly within FXU entity **321**. Although an instrumentation entity may monitor any signal within a target entity or the target entity's descendent entities, signals outside the target entity cannot be monitored.

Based upon the monitored signals within the target design entity instantiation, each instrumentation entity generates a respective set of external signals **415**, **416**, **417**. The fail,

count, harvest and phase signals within external signals **415**, **416** and **417** are connected to an instrumentation logic block **420** containing logic for recording occurrences of each of the fail, count and harvest event types. For the count events monitored in simulation model **329**, a set of counters **421** is utilized to count the number of occurrences of each count event. In a similar manner, a set of flags **424** is utilized to record the occurrence of failure events. Also, a set of counters **422** and flags **423** are combined and utilized to record the point at which a harvest event occurs and its occurrence, respectively. In one embodiment of the present invention, a cycle number is captured and stored utilizing counters **422** and flags **423** to record a harvest event.

As depicted, instrumentation logic block **420** also contains combining logic **427a-427n** for generating from the phase signals within external signals **415**, **416** and **417** what is referred to herein as "aggregate phase events." That is, each instance of combining logic **427** logically combines the individual phase events of a particular aggregate phasename and generates a respective aggregate phase event signal **429** that has a state indicating either the presence or absence in a current simulation cycle of the criteria defining a phase of operation across a region of simulation executable model **329**.

To facilitate instantiation and connection of instrumentation entities, instrumentation entity HDL source code files include a specialized comment section, hereinafter referred to as "instrumentation entity description", that indicates the target entity, the signals within the target entity to be monitored, and information specifying types of events to be monitored.

With reference now to FIG. **4C**, there is illustrated an exemplary HDL file **440** that describes instrumentation entity FXUCHK depicted in FIG. **4B**. HDL file **440** utilizes the syntax of the VHDL hardware description language. In the VHDL language, lines beginning with two dashes ("--") are recognized by a compiler as being comments. The method and system of the present invention utilize comments of a non-conventional form to indicate information about an instrumentation entity. FIG. **4C** depicts one embodiment of the present invention in which comments begin with two exclamation points in order to distinguish these comments from conventional comments in instrumentation HDL file **440**. It will be appreciated by those skilled in the art that the exemplary syntax utilized in FIG. **4C** for the provision of unconventional comments is but one of many possible formats.

Within HDL file **440**, the I/O ports of a FXUCHK entity are declared in entity declaration **450**. Within entity declaration **450**, three input ports, S_IN , Q_IN , and R_IN, respectively, are declared. Input ports, S_IN, Q_IN, and R_IN, will be attached to signal S, **374**, signal Q, **372**, and signal R, **376** respectively as described below. Input port, CLOCK, is also declared and will be connected to a signal, CLOCK, within the FXU entity. In addition, four output ports are declared: fails (0 to 1), counts (0 to 2), harvests (0 to 1) and phases (0 to 0). These output ports provide failure, count, harvest and phase signals for two failure events, three count events, two harvest events and one phase event. The names of the output ports are fixed by convention in order to provide an efficient means for automatically connecting these signals to instrumentation logic block **420**.

A set of instrumentation entity descriptors **451** is utilized to provide information about the instrumentation entity. As illustrated in FIG. **4C**, descriptor comments **451** may be categorized in a number of distinct sections: prologue and entity name declaration **452**, an input port map **453**, a set of failure

message declarations 454, a set of counter declarations 455, a set of harvest declarations 456, a set of phase declarations 490 and an epilogue 457.

The prologue and entity name 452 serve to indicate the name of the particular target entity that the instrumentation entity will monitor. Prologue and entity name declaration 452 also serves as an indication that the instrumentation entity description has begun. Specifically, the comment "--!! Begin" within prologue and entity name 452, indicates that the description of an instrumentation entity has begun. The comment "--!! Design Entity: FXU" identifies the target entity which, in HDL file 440, is design entity FXU. This declaration serves to bind the instrumentation entity to the target entity.

Input port map 453 serves as a connection between the input ports of an instrumentation entity and the signals to be monitored within the target entity. The comments begin with comment "--!! Inputs" and end with comment "--!! End Inputs". Between these comments, comments of the form "--!! inst_ent_port_name=>trgt_ent_signal_name" are utilized, one for each input port of the instrumentation entity, to indicate connections between the instrumentation entity ports and the target entity signals. The inst_ent_port_name is the name of the instrumentation entity port to be connected to the target entity signal. The trgt_ent_signal_name is the name of the signal within the target entity that will be connected to the instrumentation entity port.

In some cases a signal to be monitored lies within a descendant of a target entity. This is the case for signal S 374, which is embedded within entity C which is a descendant of entity B 326 and target FXU entity 321. Input port map 453 includes an identification string for signal S 374, which consists of the instance names of the entities within the target entity each separated by periods ("."). This identification string is prepended to the signal name. The signal mapping comment within input port map 453 for signal S 374 is therefore as follows:

    --!!S_IN=>B.C.S;

This syntax allows an instrumentation entity to connect to any signal within the target entity or the target entity's descendant entities. A signal appearing on the top level of the target design entity, has no pre-pended entity names; and therefore, has the following signal mapping comment:

    --!!R_IN=>R;

For signals on the top level of the target entity, a special connection method is provided. If the signal to be connected to has the same name as its corresponding signal in the port map of the instrumentation entity, no input port mapping comment is required and the signal will be automatically connected if no such comment is present. In other words, if the input port mapping comment is of the form:

    --!!signal=>signal;

where signal is a legal signal name without periods ("."), then the input port mapping comment is not required, and the system of the present invention will automatically make the connection. It is also possible to provide comments of the form given above to explicitly denote the signal connection. This mechanism is only operative for signals on the top level of the target entity.

Failure message declarations 454 begin with a comment of the form "--!! Fail Outputs;", and end with a comment of the form "--!! End Fail Outputs;". Each failure event output is associated with a unique event name and a failure message. This message may be output by the simulation run-time envi-

ronment upon detecting a failure event. The unique failure event name is utilized to identify the specific failure event within the model. Each failure event signal may be declared by a comment of the form "--!! n: <eventname>"failure message";" where n is an integer denoting the failure event to which the message is associated, <eventname> is the unique failure event name, and "failure message" is the message associated with the particular failure event. One and only one failure message declaration comment must be provided for each failure event monitored by the instrumentation entity.

Counter declaration comments 455 begin with a comment of the form "--!! Count Outputs;" and end with a comment of the form "--!! End Count Outputs;". Each count event output is associated with a unique variable name. This name is associated with a counter in counter logic 421 FIG. 4B. The variable name provides a means to identify and reference the particular counter associated with a particular count event. Thus, a comment of the form "--!! n: <varname> qualifying_signal [+/−];" is associated with each counter event output. Within this convention, n is an integer denoting which counter event in the instrumentation module is to be associated with a variable name "varname," and qualifying_signal is the name of a signal within a target design entity utilized to determine when to sample the count event pulse, as will be further described hereinbelow. The parameter "qualifying_signal" is followed by A+/−A to specify whether the qualifying signal will be a high active qualifying signal or a low active qualifying signal.

Harvest declarations 456 begin with a prologue comment of the form "--!! Harvest Outputs;" and end with a comment of the form "--!! End Harvest Outputs;". Each harvest event output is associated with a unique event name and a message that may be output by the simulation runtime environment when a harvest event has occurred during a simulation run. Each harvest event signal is declared in the form "--!! n: <eventname>"harvest message";" where n is an integer denoting which harvest event the message is to be associated with, <eventname> is the unique harvest event name and "harvest message" is the message to be associated with the particular harvest event. One, and only one, harvest message declaration comment must be provided for each harvest event monitored by the instrumentation entity.

Phase message declarations 490 begin with a comment of the form "--!! Phase Outputs;", and end with a comment of the form "--!! End Phase Outputs;". Each phase event output is associated with a unique event name and an aggregate phase name that tags the phase event as an input of a particular instance of combining logic 427 associated with that aggregate phase name. The unique phase event name is utilized to identify the specific aggregate phase event within the model. Each phase event signal may be declared by a comment of the form "--!! n: <eventname><phasename>;" where n is an integer denoting the bit in external signal 432 with which the phase event is associated, <eventname> is the unique phase event name, and <phasename> is the aggregate phase name with which the particular phase event is associated.

Harvest messages and event names, fail messages and event names, counter variable names, and phase event names for a simulation model are included in a simulation executable model, and lists of all the events within the model are produced in separate files at model build time. In this manner, each simulation model includes the information for each event monitored and a separate file containing this information for each event is available. Furthermore, as will be described below, the model build process names each event

within the model (count, fail, harvest and phase) in such a manner as to insure that each event has a unique name with certain useful properties.

Finally, epilogue comment **457** consists of a single comment of the form "--!! End;", indicating the end of descriptor comments **451**. The remainder of instrumentation entity HDL file **440** that follows the I/O declarations described above is an entity body section **458**. In entity body section **458**, conventional HDL syntax is utilized to define internal instrumentation logic necessary to detect the various events on the input port signals and convey these events to the output port signals.

In addition to descriptor comments **451**, that are located in the HDL source code file for an instrumentation entity, an additional comment line is required in the target entity HDL file. A comment of the form "--!! Instrumentation: name.vhdl", where name.vhdl is the name of the instrumentation entity HDL file, is added to the target entity HDL source code file. This comment provides a linkage between the instrumentation entity and its target entity. It is possible to have more than one such comment in a target entity when more than one instrumentation entity is associated with the target entity. These HDL file comments will hereinafter be referred to as "instrumentation entity instantiations".

With reference now to FIG. **4D**, there is depicted a model build process implemented by program code in accordance with the teachings of the present invention. For ease of implementation, the program code that performs the illustrated model build process is depicted as including HDL compiler **462**, an instrumentation load tool **464**, and a model build tool **446**, referred to herein collectively as "model build program code". As noted above, processing of the model build program code may be advantageously controlled by a control program **46**. In the model build process, instrumentation load tool **464** is utilized to alter the in-memory proto and instance data structures of a simulation model, thereby adding instrumentation entities to the simulation model. Instrumentation load tool **464** utilizes descriptor comments **451** within instrumentation HDL files **461** to create instance data structures for the instrumentation entities within a simulation model.

The model build process of FIG. **4D** begins with design entity HDL files **340** and, potentially, one or more design entity proto files **345** (available from a previous run of HDL compiler **462**), instrumentation entity HDL files **460**, and potentially, one or more instrumentation entity proto files **461** (available from a previous run of HDL compiler **462**) and an optional phase file **492**.

In one preferred embodiment, phase file **492** lists the legal aggregate phasenames for the simulation model and, for each named phase, the Boolean operator (e.g., AND, OR, etc.) of the combining logic **427** that combines the phase event(s), if any, of that phase from each instrumentation entity to obtain an aggregate phase event signal **429**. Thus, an entry within phase file **492** may take the form: <phasename>, <combining function>. It is also preferred if a special "end-of-testcase" (EOT) phasename is predefined, such that detection of a model-wide EOT phase can be utilized to halt execution of a testcase on a hardware accelerator, as discussed further below with respect to FIG. **5B**.

Although not required, it is also preferable if each entry within phase file **492** also supports the definition of an associated region of the simulation model in which the specified phase name is valid. In this embodiment, an exemplary entry within phase file **492** may take the form: <phasename>, <combining function>, <instance>, where <instance> is an optional parameter specifying the design entity instance defining the region of the simulation model in which the specified aggregate phasename is valid. HDL compiler **462** is

also preferably configured to interpret entries in phase file **492** having no specified <instance> parameter as being associated with the top-level design entity instance of the simulation model. Associating aggregate phasenames with particular regions of the simulation model in this manner permits HDL files from different projects to be synthesized into a single simulation model without concern for conflicting aggregate phasenames.

In a preferred embodiment, the predefined EOT phasename is not subject to regional definition, ensuring that the EOT phasename has the same name across all subsets of a simulation executable model and that no phasename conflict will arise for the EOT phasename when multiple smaller simulation models are combined to form a simulation executable model of a larger system. Because the EOT phasename is predefined, it is also unnecessary and preferably an error to explicitly enumerate the EOT phasename in a phase file **492**.

HDL compiler **462** processes design entity HDL files **340** and instrumentation entity HDL files **460** following an augmentation of algorithm process_HDL_file( ) that provides for efficient incremental compilation of the design and instrumentation entities comprising a simulation model. HDL compiler **462** loads proto data structures from design entity proto files **345** and instrumentation entity protos files **460**, if such proto files are available and consistent. If such proto files are not available or are not consistent, HDL compiler **462** compiles design entity HDL files **340** and instrumentation entity HDL files **460** in order to produce design entity proto files **344** and instrumentation entity proto files **468**. (Design entity proto files **344** and instrumentation entity proto files **468** are available to serve as design entity proto files **345** and instrumentation entity proto files **460** respectively for a subsequent run of HDL compiler **462**.)

HDL compiler **462** processes instrumentation entity HDL files **461**, instrumentation entity proto files **460**, design entity HDL files **340** and design entity proto files **345** by reference to phase file **492**. In particular, HDL compiler **462** verifies that any <instance> parameter of an entry within phase file **492** has a corresponding design entity instance within design entity HDL files **340** or design entity proto files **345** and generates an error otherwise. In addition, HDL compiler **462** verifies that any aggregate phasename referenced within instrumentation entity HDL files **461** or instrumentation entity proto files **460** has a corresponding <phasename> parameter within an entry in phase file **492** and generates an error otherwise. No error is generated by HDL compiler **462** if aggregate phasenames specified within phase file **492** are unused within instrumentation entity HDL files **461** or instrumentation entity proto files **460**.

In addition, HDL compiler **462** creates in-memory design proto data structures **463** and design instance data structures **465** for the design entities of a simulation model. HDL compiler **462** also creates in-memory instrumentation proto data structures **466** for the instrumentation entities of a simulation model.

In order to minimize processing overhead HDL compiler **462** neither reads nor processes descriptor comments **451**. However, HDL compiler **462** does recognize instrumentation entity instantiation comments within target entity HDL files. As such, HDL compiler **462** cannot create instance data structures instrumentation entity data structures **467**. The creation of instance data structures requires interconnection information contained within descriptor comments **451** not processed by HDL compiler **462**. HDL compiler **462** does, however, create instrumentation proto data structures **466**.

The in-memory design proto data structures **463**, design instance data structures **465**, and instrumentation entity proto

data structures **466**, are processed by instrumentation load tool **464**. Instrumentation load tool **464** examines design entity proto data structures **463** and design entity instance data structures **465** to determine those design entities that are target entities. This examination is accomplished by utilizing a particular comment format as previously described.

All target entities that are loaded from design entity proto files **345** contain an instantiation for any associated instrumentation entity. Therefore, instrumentation load tool **464** merely creates an instance data structure **467** for any such instrumentation entity and passes, the unaltered design proto data structure **463** to instrumented design proto data structure **463a**, and passes design instance data structure **465** to instrumented design instance data structure **465a**.

If however, a target entity is loaded from design entity HDL files **340**, rather than from design entity proto files **345**, instrumentation load tool **464** must alter its design proto data structure **463** and its design instance data structure **465** to instantiate an associated instrumentation entity. An instrumented design proto data structure **463a** and instrumented design instance data structure **465a** are thereby produced. In addition, instrumentation load tool **464** creates an instrumentation instance data structure **467** for each instrumentation entity associated with the current design entity.

The design entity proto data structures **463** that are altered by instrumentation load tool **464** are saved to disk **33** of computer system **10** as design entity proto files **344**. Design entity proto files **344**, which may include references to instrumentation entities, are directly loaded by a subsequent compilation of a simulation model, thus saving processing by instrumentation load tool **464** on subsequent recompilations unless an alteration is made to a design entity or an associated instrumentation entity.

In order for HDL compiler **462** to determine if alterations were made to either a target design entity or the target design entity's associated instrumentation entities, the BOM of a target design entity is expanded to include the HDL files constituting the instrumentation entities. In this manner, HDL compiler **462** can determine, by inspection of the BOM for a given design entity, whether to recompile the design entity and the design entity's associated instrumentation entities or load these structures from proto files **345** and **461**.

Instrumentation load tool **464** creates a unique proto and instance data structure for instrumentation logic block **420** and connects the fail, harvest, count and phase event signals from each instrumentation entity instantiation to instrumentation logic block **420**. Model build tool **446** processes in-memory proto and instance data structures **463a**, **465a**, **467**, **466** to produce instrumented simulation executable model **480**.

In HDL compiler **462**, algorithm process_HDL_file( ) is augmented to allow for the incremental compilation of design and instrumentation entities. A pseudocode implementation of a main control loop of HDL compiler **462** is shown below:

```
process_HDL_file2(file,design_flag)                              5
{                                                               10
    if (NOT proto_loaded(file)) {                              15
        if (exists_proto_file(file) AND check_bom(file)) {     20
            load_proto(file);                                  25
        }else {                                                30
            parse_HDL_file(file)                               35
            for (all instances in file) {                      40
                process_HDL_file2(instance, design_flag);      45
            }                                                  50
            if (design_flag=TRUE) {                            55
```

-continued

```
                for (all instrumentation instances in file){   60
                    process_HDL_file2(instance, FALSE);        65
                }                                              70
            }                                                  75
            create_proto(file);                                80
            write_proto_file(file);                            90
        }                                                      95
    }                                                         100
    if (design_flag=TRUE) {                                   105
        create_instance(file);                                110
    }                                                         115
}                                                             120
```

Algorithm process_HDL_file2( ) is an augmentation to process_HDL_file( ) of HDL compiler **342** in order to support the creation of instrumented simulation models. The algorithm is invoked with the name of the top level design entity passed through parameter file and a flag indicating whether the entity being processed is a design entity or an instrumentation entity passed through parameter design_flag (design_flag=TRUE for design entities and FALSE for instrumentation entities). Algorithm process_HDL_file2( ) (line 5) first checks, by means of routine proto_loaded( ) (line 15), if the proto for the current entity is already present in memory **44**. If so, processing passes to line 105. Otherwise, control is passed to line 20 and 25 where disk **33** of computer system **10** is examined to determine if proto files for the entity and its descendants (including instrumentation entities, if any) exist and are consistent. If so, the appropriate proto files are loaded from disk **10** by routine load_proto( ) (line 25) creating proto data structures, as necessary, in memory **44** for the current entity and the current entity's descendants including instrumentation entities.

If the proto file is unavailable or inconsistent, control passes to line 35 where the current entity HDL file is parsed. For any entities instantiated within the current entity, lines 40 to 55 recursively call process_HDL_file2( ) (line 5) in order to process these descendants of the current entity. Control then passes to line 55 where the design_flag parameter is examined to determine if the current entity being processed is a design entity or an instrumentation entity. If the current entity is an instrumentation entity, control passes to line 80. Otherwise, the current entity is a design entity and lines 60 to 70 recursively call process_HDL_file2( ) (line 5) to process any instrumentation entities instantiated by means of instrumentation instantiation comments. It should be noted that algorithm process_HDL_file2( ) (line 5) does not allow for instrumentation entities to monitor instrumentation entities. Any instrumentation entity instantiation comments within an instrumentation entity are ignored. Control then passes to line 80 where proto data structures are created in memory **44** as needed for the current entity and any instrumentation entities. Control then passes to line 90 where the newly created proto data structures are written, as needed to disk **33** of computer system **10**.

Control finally passes to line 105 and 110 where, if the current entity is a design entity, instance data structures are created as needed for the current entity and the current entity's descendants. If the current entity is an instrumentation entity, routine create_instance( ) (line 110) is not called. Instrumentation load tool **464** is utilized to create the in-memory instance data structures for instrumentation entities.

It will be apparent to those skilled in the art that HDL compiler **462** provides for an efficient incremental compilation of design and instrumentation entities. It should also be noted that the above description is but one of many possible

means for accomplishing an incremental compilation of instrumentation entities. In particular, although many other options also exist, much, if not all, of the functionality of instrumentation load tool 464 can be merged into HDL compiler 462.

With reference now to FIG. 4E wherein is shown a depiction of memory 44 at the completion of compilation of simulation model 329 with instrumentation entities FXUCHK and FPUCHK. Memory 44 contains proto data structures 481, one for each of the design and instrumentation entities referred to in simulation model 329. In addition, design and instrumentation instances in simulation model 329 are represented by instance data structures 482. The instance data structures are connected by means of pointers indicating the hierarchical nature of the instantiations of the design and instrumentation entities within simulation model 329.

With reference now to FIG. 5A, failure flags 424 of instrumentation logic block 420 are depicted in greater detail. Failure flags 424 comprise registers 500a-500n utilized to accept and store an indication of the occurrence of a failure event. In what follows, the operation of a single failure flag for a particular failure event 502 will be discussed. The operation of all failure flags is similar.

Register 500a holds a value that represents whether a failure event 502 has occurred or not. Register 500a is initially set to a value of '0' by the simulation run-time environment at the beginning of a simulation run. When failure event 502, if enabled at register 507a, occurs, register 500a is set to a value of a logical '1', thereby indicating the occurrence of a failure event. Register 500a is driven by logical OR gate 501. Logical OR gate 501 performs a logical OR of the output of register 500a and a qualified failure signal 503 to create the next cycle value for register 500a. In this manner, once register 500a is set to a logical '1' by the occurrence of an enabled failure event, register 500a maintains the value of a logical '1' until reset by the simulation runtime environment. Likewise, register 500a maintains a value of '0' from the beginning of the simulation run until the occurrence of the failure event, if enabled.

Qualified failure signal 503 is driven by logical AND gate 505. Logical AND gate 505 produces, on qualified failure signal 503, the logical AND of failure signal 506 and the logical NOT of register 507a. Register 507a serves as an enabling control for qualified failure signal 503. If register 507a contains a value of '0', logical AND gate 505 will pass failure event signal 506 unaltered to qualified failure signal 503. In this manner, the monitoring of the failure event is enabled. Registers 507a-507n are set, by default, to a value of '0'. However, if register 507a contains a value of a logical '1', qualified failure signal 503 will remain at a value of '0' irrespective of the value of failure event signal 506, thereby disabling the monitoring of failure event 502. In this manner, register 508, consisting of registers 507a-507n, can mask the occurrence of any subset of failure events in the overall simulation model from registers 500a-500n.

To efficiently implement the ability to selectively disable the monitoring of failure events, the simulation run-time environment includes a function that allows a user to disable monitoring of a specific failure event for a given instrumentation entity. This function will automatically set the appropriate registers among registers 507a-507n within register 508 to disable the monitoring of a particular failure event for every instance of the instrumentation entity within the overall simulation model. Instrumentation load tool 464 and model build tool 446 encode sufficient information within instru-

mented simulation executable model 480 to determine which failure bits within register 508 correspond to which instrumentation entities.

The ability to selectively disable monitoring of failure events is of particular use in large batch-simulation environments. Typically, in such an environment, a large number of general purpose computers, running software or hardware simulators, are dedicated to automatically running a large number of simulation runs. If a simulation model with a faulty instrumentation entity that incorrectly indicates failure events is run in such an environment, a large number of erroneous failures will be generated causing lost time. By selectively disabling failure events within instrumentation entities, the present invention allows simulation to continue while only disabling erroneous failure signals rather than having to disable all failure monitoring. This option is particularly useful when the process of correcting a faulty instrumentation entity and creating a new simulation model is substantially time consuming. The present invention also provides similar enabling and disabling structures for the harvest and count events within a model.

Logical OR gate 512 is utilized to produce a signal 511, which indicates whether any failure event within the model has occurred. This signal is utilized to allow hardware simulators to efficiently simulate simulation models that have been instrumented according to the teachings of the present invention.

It will further be appreciated upon reference to FIG. 5A that the illustrated technique for disabling particular failure events can also be applied to disable other types of events, such as phase events.

With reference now to FIG. 5B there is illustrated in greater detail, features of the present invention utilized to support efficient execution of an instrumented simulation model on a hardware simulator. It should be noted that for most hardware simulators, the operation of polling a facility within a simulation model during a simulation run is often a time consuming operation. In fact, if facilities must be polled every cycle, it is often the case that as much time, if not considerably more, is spent polling a simulation model for results rather than running the actual simulation. As such, it is advantageous when using a hardware simulator to avoid polling facilities within the model during a simulation run. In addition, many hardware simulators provide a facility that instructs the hardware simulator to run a simulation without interruption until a specific signal within the simulation model attains a specific value. This facility usually results in the highest performance for a simulation run on a hardware simulator.

In order to execute simulation model 520 on a hardware simulator, one or more instances of instrumentation entities, such as CHK1 550a and CHKN 550n, are instantiated in simulation executable model 520. As indicated, each instrumentation entity instantiation 550 generates from one or more input signals a set of external signals including an external phase signal 432 of one or more bits. At least one bit among external phase signals 432 can be defined as having an "end-of-testcase" (EOT) phasename and is employed to signal occurrence of EOT phase events. HDL compiler 462 also instantiates end-of-testcase combining logic 427n (in this case, an AND gate 552) within the instrumentation logic block 420 of simulation model 520 to implement the combining function predefined for the EOT phasename for simulation model 520. The assertion of the output of such end-of-testcase combining logic 427n, that is, termination signal 513 output by AND gate 552, thus indicates that the end of a testcase has been reached. Without termination signal 513, it would be necessary to directly poll facilities within simula-

tion model **520** to determine when a simulation run is complete. Other phase events reported within external phase signals **432** are combined by combining logic **427a-427m** to generate aggregate phase event signals **429a-429m** within instrumentation logic block **420**, as described above.

To efficiently locate and diagnose problems in simulation model **520**, it is also advantageous to allow a simulation run to be stopped immediately whenever a failure event occurs during simulation of simulation model **520**. (Harvest events and count events are typically only polled at the end of a simulation run.) Stopping simulation immediately upon occurrence of a failure event allows a user to easily locate the failure event within the simulation run, thereby facilitating debugging of the failure. If an EOT phasename is predefined for a simulation executable model **520**, a logical OR gate **515** will automatically be included within the simulation model **520**. Logical OR gate **515** receives as inputs termination signal **513** and the failure signal **511** generated by OR gate **512** and produces as an output the logical OR of signals **511** and **513** on signal **516**. Signal **516** is therefore asserted to a logical '1' whenever the simulation run has completed (signal **513** high) or a failure event has occurred (signal **511** high). In the alternative, if an EOT phasename is not predefined for simulation executable model **520**, logical OR gate **515** is not included in the model and failure signal **511** is directly connected to signal **516**. The name of signal **516** is fixed to a particular name by convention.

While the above described system and method provides a practical means of instrumenting simulation models, in certain circumstances additional techniques may be used in order to enhance the ease with which a user may instrument a simulation model. In design, it often occurs that there are common design or instrumentation logic constructs that are often repeated and possess a regular structure.

By utilizing knowledge of the regular structure of these design and instrumentation logic constructs, it is often possible to define a syntax that describes the instrumentation logic with considerably greater efficiency than would be possible utilizing a conventional HDL construct. By utilizing this syntax as an unconventional HDL comment within a design VHDL file, it is possible to create instrumentation entities with considerably greater ease and efficiency.

Such comments within a design entity will be referred to hereinbelow as an embedded instrumentation entity comment while the instrumentation logic created by such a comment will be referred to as an embedded instrumentation entity.

A common logic design construct is the so-called "finite state machine". A finite state machine typically consists of a number of storage elements to maintain the "state" of the state machine and combinatorial logic that produces the next state of the state machine and its outputs. These constructs occur with great frequency in typical logic designs and it is advantageous to be able to efficiently instrument these constructs.

A typical set of count and failure events for a finite state machine includes counting the number of times a state machine cycles from a given current state to some next state, counting the number of functional cycles the state machine spends in each state, ensuring that the state machine does not enter an illegal state, and ensuring that the state machine does not proceed from a current given state to an illegal next state. This list of events is but one of many possible sets of events that can be used to characterize a finite state machine and is used in an illustrative manner only.

With reference now to FIG. **6** there is depicted a representation of an exemplary state machine **800**. Exemplary state machine **800** consists of five states, labeled S**0**, S**1**, S**2**, S**3**, and S**4** respectively, and nine legal state transitions between

these states. In what follows, it is assumed that state machine **800** consists of three latches and a set of combinatorial logic to produce the next state function. It is further assumed that the states are encoded into the three latches following the usual and customary encoding for integers. That is to say, state S**0** gets an encoding of 000_{bin}, state S**1** gets an encoding of 001_{bin}, state S**2** gets and encoding of 010_{bin}, and so on.

With reference now to FIG. **7** there is shown an exemplary design entity **850** referred to as entity FSM with instance name FSM, which contains one instance of state machine **800**. Furthermore, a signal output **801**, "fsm_state (0 to 2)", contains a three bit signal directly connected to the outputs of the three storage elements comprising the state elements of state machine **800**. A signal input **802**, fsm_clock, applies a clocking signal that controls the storage elements for state machine **800**.

In order to instrument state machine **800**, it would conventionally be necessary to create an instrumentation entity VHDL file containing the logic necessary to detect the desired state machine events and pass them through to count and fail events. Such an instrumentation entity file with appropriate instrumentation entity descriptor comments would typically require substantially more lines of code than the HDL description of the state machine itself. Such a circumstance is undesirable. However, in the case of a regular logic structure such as a finite state machine, it is possible to define a brief syntax that characterizes the finite state machine without resorting to a separate instrumentation VHDL entity.

With reference now to FIG. **8** there is illustrated an exemplary HDL file **860** for generating design entity **850** with an embedded instrumentation entity for monitoring the behavior of FSM **800**. Specifically, an embedded instrumentation entity comment **852** is illustrated. As depicted in FIG. **8**, embedded instrumentation entity comment **852** comprises a number of distinct sections including: a prologue and embedded instrumentation name declaration **853**, a state machine clock declaration **859**, a state element declaration **854**, a state naming declaration **855**, a state element encoding declaration **856**, a state machine arc declaration **857**, and an epilogue **858**.

Prologue and embedded instrumentation entity name declaration comment **853** serves to declare a name that is associated with this embedded instrumentation entity. This comment line also serves to delineate the beginning of an embedded instrumentation entity comment sequence.

As further depicted in FIG. **8**, declaration comment **853** assumes a non-conventional syntax of the form: "--!! Embedded TYPE: name", wherein "--!! Embedded" serves to declare an embedded instrumentation entity, "TYPE" declares the type of the embedded instrumentation entity B FSM in this case, and "name" is the name associated with this embedded instrumentation entity.

State machine clock declaration comment **859** is utilized to define a signal that is the clocking control for the finite state machine.

State element declaration comment **854** is utilized to specify the state-machine state storage elements. This comment declares the storage elements or signal names that constitute the state-machine state. In state machine **800**, the signals fsm_state (0 to 2) constitute the state machine state information.

State naming declaration comment **855** is utilized to declare labels to associate with various states of the given state machine. These labels are utilized in state machine arc declaration comment **857** when defining the legal state transitions within the given state machine.

State element encoding declaration comment **856** is utilized to define a correspondence between the state machine

US 7,493,248 B2

23

labels defined by state naming declaration comment **855** and the facilities declared by state element declaration comment **854**. In the example shown, the labels of comment **855** are associated by position with the encodings given in comment **856** (i.e., the state labeled "S0" has the encoding 000_{bin}, the state labeled "S1" has the encoding 001_{bin}, etc.).

State-machine arc declaration comment **857** defines the legal state transitions within the state machine. The various transitions of the state machine are given by terms of the form "X=>Y" where X and Y are state machine state labels given by comment **855** and X represents a previous state machine state and Y a subsequent state machine state.

Epilogue comment **858** serves to close the embedded instrumentation entity comment. The specific syntax and nature of the comments between the prologue and embedded instrumentation name declaration and the epilogue will vary with the specific needs of the type of embedded instrumentation entity being declared.

Embedded instrumentation entity comment **852** is inserted within the VHDL file of the design entity that contains the finite state machine in question. The embedding of instrumentation for finite state machine **800** is made possible by the non-conventional comment syntax illustrated in FIG. **8** and is substantially more concise than a conventional HDL instrumentation entity suitable for accomplishing the same function.

Utilizing such embedded non-conventional comments, the system of the present invention creates an instrumentation entity, as described below, for instrumenting the state machine without the need to resort to creating a separate HDL file instrumentation entity.

To support compilation and creation of embedded instrumentation entities, the previously described compilation process of FIG. **4D** is enhanced as described herein. First, HDL compiler **462** is altered to recognize the presence of embedded instrumentation entity comments. If, during compilation of a design HDL file, and subject to the constraints described above for incremental compilation, HDL compiler **462** detects one or more embedded instrumentation entity comments within the source code file, HDL compiler **462** places a special marker into design entity proto data structure **463**.

When instrumentation load tool **464** is passed control, proto data structures **463** are searched in order to locate the special marker placed by HDL compiler **462** indicating embedded instrumentation entity comments. Such protos represent the design HDL files with embedded instrumentation entities that have been re-compiled in the current compilation cycle.

When instrumentation load tool **464** locates a proto data structure **463** with the special marker, the corresponding VHDL source code file for the design entity is opened and parsed to locate the one or more embedded instrumentation entity comments. For each of these comments, instrumentation load tool **464** creates a specially named proto data structure **463a**, and further generates a corresponding instance data structure **465a** that is instantiated within the design entity. In addition, instrumentation load tool **464** removes the special marker inserted by HDL compiler **462** to prevent unnecessary re-instrumentation of the design proto on subsequent re-compiles.

Within these created embedded instrumentation entity protos, instrumentation load tool **464** directly creates the necessary instrumentation logic required by the embedded instrumentation entity without the need for a VHDL file to specify this instrumentation and connects this logic to instrumentation logic block **420** of FIG. **4D**. The updated design proto along with the embedded instrumentation entity proto and

24

instance data structure are saved to disk and serve as inputs to subsequent compiles, removing the need to produce embedded instrumentation entities on subsequent recompiles.

With reference now to FIG. **9**, design entity **850** is shown instrumented with embedded instrumentation entity **900**. Embedded instrumentation entity **900** is created as a proto instantiated within design entity **850** wherein the embedded non-conventional instrumentation entity comment occurs. The embedded instrumentation entity thus may be replicated automatically within an overall design wherever the specific design entity is instantiated.

Embedded instrumentation entity **900** is named in a unique manner based on the name associated with the embedded instrumentation entity by the prologue and embedded instrumentation name declaration comment. This name is prepended with a special character (shown as a "$" in FIG. **9**) that is not a recognized naming convention for the platform HDL. In this manner, the names of the embedded instrumentation entities cannot conflict with the names of any other design or standard instrumentation entities.

Furthermore, the names associated with the various events defined by the embedded instrumentation entity (the "varname" for the count events, for example) are also derived in a fixed manner from the name associated with the embedded instrumentation entity. The user is required to ensure that the names of embedded instrumentation entity events do not conflict with the names of standard instrumentation entity events and further than the names of the embedded instrumentation entities within a given design do not themselves conflict.

It should also be noted that if a design entity contains more than one embedded instrumentation entity, the embedding process described with reference to FIG. **8** and FIG. **9** is simply repeated for each such instrumentation entity. In addition, since the protos for the embedded instrumentation entities are created at the same time as the design protos itself, no changes to the BOM mechanism used for incremental compiles are required. The protos for the embedded instrumentation entities can be considered, for purposes of incremental compilations, to be mere extensions to the design proto itself.

The present invention discloses a method and system for naming events within a simulation model that prevents name collisions between events in different instrumentation entities, allows for the arbitrary re-use of components of a model in models of arbitrarily increasing size, and furthermore allows for processing designated events in a hierarchical or non-hierarchical manner.

When all instances of an event are considered as a whole without regard to specific instances, the event is considered in a "non-hierarchical" sense. Likewise, when an event is considered with regard to each and every instance, it is considered in a "hierarchical" sense. When considering count events, for example, it is often convenient to track the number of times a particular count event occurred in the aggregate without concern to exactly how many times the count event occurred in each particular instance within a simulation model.

Each type of event: count, fail, harvest and phase, is given a separate event namespace by construction. Each event class is therefore an independent group preventing naming collisions between the event types. The data structure of the present invention is independently applied to each of the different event types to ensure correctness within each event class.

In the embodiments illustrated in FIGS. **10A**, **10B**, **10C**, and **10D**, the system and method of the present invention are described with respect to count events. One skilled in the art

will appreciate and understand the extensions necessary to apply the same techniques to other event classes such as failures, harvests and phases.

With reference to FIG. **10A**, there is depicted a block diagram representation of simulation model **1000** containing a number of design and instrumentation entities. As illustrated in FIG. **10A**, simulation model **1000** includes two instances of a design entity X, with instance names X**1** and X**2** respectively. Within each of design entity instances X**1** and X**2** is instantiated an instance of an instrumentation entity B**3**, **1012***a* and **1012***b*. Design entity instances X**1** and X**2** further comprise instances, **1014***a* and **1014***b*, respectively, of design entity Z which further contains instances, **1016***a* and **1016***b*, of instrumentation entity B**1** and instances, **1018***a* and **1018***b*, of instrumentation entity B**2**.

Finally, simulation model **1000** includes an instance of design entity Y, with instance name Y, containing an instance of instrumentation entity B**4** **1022**. Design entity instance Y contains an instance, **1024**, of design entity Z with further instances, **1016***c* and **1018***c*, of instrumentation entities B**1** and B**2** respectively.

In what follows the methods of the present invention for uniquely naming events will be considered in the context of exemplary model **1000**. It will be assumed in the following description that each instrumentation entity (B**1**, B**2**, B**3**, and B**4**) has declared a single count event with event name "count1".

In accordance with the method and system of the present invention, the user must uniquely name each type of event (count, fail, harvest or phase) within a specific instrumentation entity, i.e., the user cannot declare any two events of the same type within the same instrumentation entity with the same event name. Such a constraint does not conflict with the stated goals of the present invention in that a given instrumentation entity is usually created by a specific person at a specific point in time, and maintaining unique names within such a limited circumstance presents only a moderate burden to the user. The data structure disclosed herein does, however, prevent all name collisions between events in different instrumentation entities, and allows for processing the events in a hierarchical and/or non-hierarchical manner.

As previously explained, an HDL naming convention must uniquely identify all the entities within a given design. This constraint is inherent to HDLs and applies to design entities as well as instrumentation entities. In accordance with conventional VHDL entity naming constructs, it is technically possible for two design entities to share the same entity name, entity_name. However, such identically named entities must be encapsulated within a VHDL library from which a valid VHDL model may be constructed. In such a circumstance, entity_name, as it is utilized herein, is equivalent to the VHDL library name concatenated by a period (".") to the entity name as declared in the entity declaration.

Pre-pending a distinct VHDL library name to the entity name disambiguates entities sharing the same entity name. Most HDLs include a mechanism such as this for uniquely naming each design entity. Design entities must be unambiguously named in order to determine which particular entity is called for in any given instance in a simulation model. The present invention employs the prevailing naming mechanism of the native HDL to assign unique entity names for design entities throughout a given model and leverages the uniqueness property of entity names and the uniqueness of each instance's instantiation identifier to create an "extended event identifier" for each event within the simulation model.

With reference to FIG. **10B**, there is illustrated a representation of the fields in an extended event identifier data struc-

ture, alternatively referred to herein as an "event list", in accordance with one embodiment of the present invention. The extended event identifier begins with instantiation identifier field **1030**. This field, as described hereinbefore, consists of the instance identifiers, proceeding from the top level entity to the direct ancestor of the given instance within the simulation model separated by periods ("."). This string is unique for each and every instance of the event within the model. The extended event identifier further includes an instrumentation entity field **1032**, a design entity field **1034**, and an eventname field **1036**.

Instrumentation entity field **1032** contains the name of the instrumentation entity (or the name assigned to an embedded instrumentation entity) that generates the simulation event. Design entity field **1034** contains the entity name of the design entity in which the event occurs. Eventname field **1036** is the name given to the event in the instrumentation entity description comments of an instrumentation entity or the event name assigned to an event within an embedded instrumentation entity. These four namespace fields comprise a unique identifier for each event within a simulation model that allows for the re-use of components within other models without risk of name collisions and the consideration of events in a hierarchical or non-hierarchical sense.

With reference now to FIG. **10C**, there is shown a list of extended event identifiers for model **1000**. Event identifiers **1040**, **1041**, **1042**, **1043**, **1044**, **1045**, **1046**, **1047**, and **1048** are declared within simulation model **1000** to designate count events having eventname "count1". The extended event identification procedure of the present invention will be described in the context of these extended event identifiers.

The uniqueness of the names in design entity name field **1034** is a primary distinguishing factor between events. By including the design entity name in the extended event identifier, each design entity is, in effect, given a unique namespace for the events associated with that design entity, i.e., events within a given design entity cannot have name collisions with events associated with other design entities.

It is still possible however, to have name collisions between events defined by different instrumentation entities that are incorporated within a single design entity. Events **1041** and **1042**, for example, if identified solely by the design entity name, have a name collision. Both are events with eventname "count1" within design entity Z, and if labeled as such, are indistinguishable. In order to alleviate a naming collision between events **1041** and **1042**, the present invention employs instrumentation entity field **1032**. By referencing the design entity and instrumentation entity names, both of which are unique with respect to themselves and each other, a unique event namespace is created for each instrumentation entity associated with any given design entity. For example, event identifier **1041** and **1042** would be in conflict (both named Z.count1), unless the respective instrumentation entity names are included within the extended event identifier to produce names B**1**.Z.count1 and B**2**.Z.count2 for these events.

It should be noted that it is possible to uniquely name each event by using instrumentation entity name field **1032** alone. Due to the uniqueness property of instrumentation entity names, event names that are only named by the instrumentation entity name and the event name field will be necessarily unique.

However, such a naming scheme is insufficient for associating events with a given design entity. In practice, it is desirable to associate events with the design entity in which they occur rather than associating them with the potentially numerous instrumentation entities that are utilized to track them. Moreover, referencing the appropriate design entity

within the eventname allows all the events associated with a given design entity to be centrally referenced without the need to ascertain the names of all the instrumentation entities associated with the given design entity. The data structure of the present invention utilizes both the instrumentation entity and design entity names in naming events for ease of reference at the cost of moderate uniqueness redundancy in the event names.

In an alternative embodiment of the present invention, the instrumentation entity name is not included within the extended event identifier. Referring to FIG. 10D, such an alternative extended event identification data structure is depicted. As shown in FIG. 10D, events are named by instantiation identifier field **1030**, design entity name field **1034**, and event name field **1036**.

Such a data structure provides name collision protection between design entities but not within design entities. That is, the user must ensure that events names for events associated with a given design entity do not collide. In case of user error in this regard, model build tools may be utilized to detect an event name collision condition during model compilation. The alternative data structure depicted in FIG. 10D provides for simpler naming and referencing of events at the expense of requiring the user to prevent name collisions for events associated with a given design entity.

Returning to FIG. 10B, the combination of instrumentation entity field **1032**, design entity name field **1034**, and event-name field **1036** for a given event, provides a unique identifier for any given event without regard to multiple instantiations of the event. In order to uniquely distinguish between multiple instantiations of an event, instantiation identifier field **1030** is included in the extended event identifier. Instantiation identifier field **1030** field, by its construction, provides a unique string for any instance of an entity within any simulation model.

When evaluating occurrences of an event in a non-hierarchical sense, instantiation identifier field **1030** is ignored while searching for matching events. As illustrated in FIG. 10C, for example, a non-hierarchical query for the number of time a "count1" event occurs within design entity Z as detected by instrumentation entity B1, utilizes the following list of count eventnames:

X1.Z	B1	Z	COUNT1
X2.Z	B1	Z	COUNT1
Y.Z	B1	Z	COUNT1.

These count events are added together to form an aggregate count of the total number of time the specific event occurred within the simulation model.

A hierarchical query includes specific criteria to match against the hierarchy field to limit the counter or counters found to specific instances of the requested event. For example, a query to obtain the count1 event of instrumentation entity B1 within the X1.Z instance of design entity Z utilizes the following count eventname:

X1.Z	B1	Z	COUNT1,

which represents the number of times the count1 event was counted by instrumentation entity B1 within design entity instance X1.Z for a particular simulation interval.

By providing matching model hierarchy criteria against instantiation identifier field **1030**, it is possible to consider the events with respect to their particular instance or instances within the model, i.e., a hierarchical query. A non-hierarchical query merely ignores the hierarchy field and returns all the instances of the requested events within the model.

With reference to FIG. 11A, there is depicted a block diagram illustrating a simulation model **1100** in which the hierarchical event processing of the present invention is applicable. Simulation model **1100** comprises a top-level design entity **1130** in which a pair of lower-level design entities **1102** and **1120** is instantiated. A design entity **1104** containing instrumentation entity **1106** is included within design entity **1102**. As illustrated in FIG. 11A, instrumentation entity **1106** includes logic **1110** for generating a simulation event **1108** from signal set **1132** from within design entity **1104**. Design entity **1120** includes an instrumentation entity **1122** that generates a simulation event **1124** using signal set **1134**.

Utilizing the techniques described hereinbefore, generating a hierarchical event that is some logical combination of events **1108** and **1124** requires the creation of an instrumentation entity associated with top level design entity **1130** that references signal sets **1132** and **1134**. Conventionally, such an instrumentation entity would substantially reproduce instrumentation logic **1110** and **1126** to process signal sets **1132** and **1134**, respectively, thus producing a copy of events **1108** and **1124**. Such a procedure is inefficient and prone to error. If, for example, changes are made to any or all of signal sets **1132** and **1134**, or instrumentation logic **1110** and **1126**, these changes would have to be accurately repeated in the instrumentation entity logic for the combined event.

The present invention provides a mechanism whereby events, such as events **1108** and **1124**, are directly referenced and utilized as inputs to cross-hierarchical instrumentation entities. In this manner, signal connections **1132** and **1134**, as well as instrumentation logic **1110** and **1126**, are directly re-utilized to produce the desired hierarchical event.

To facilitate direct referencing of events within simulation models, a specialized data structure is implemented within instrumentation entity input port map comment syntax. This data structure directly connects input ports of instrumentation entities to cross-hierarchical events within a simulation model.

For the embodiment depicted in FIG. 11A, an instrumentation entity **1150** is instantiated within top-level design entity **1130** to generate a hierarchical event **1156** that is some function of events **1108** and **1124**. As illustrated in FIG. 11A, instrumentation entity **1150** includes a pair of inputs **1151** and **1152** that are directly connected to events **1124** and **1108**, respectively, utilizing the augmented syntax described below. These input connections are logically combined using instrumentation logic **1154** to produce a cross-hierarchical event **1156**.

With reference to FIG. 11B, there is depicted a set of input port mapping comments for performing cross-hierarchical processing of simulation model events in accordance with the teachings of the present invention. In what follows, it is assumed that events **1108** and **1124** are count events with event names event_**1108** and event_**1124**, respectively, and that these events are connected to input ports event_**1108**_in and event_**1124**_in on instrumentation entity **1150**. As depicted in FIG. 11B, a first input port mapping comment **1161** contains data for referencing event **1108** to input port event_**1108**_in. A second input port mapping comment **1162** contains data for referencing event **1124** to input port event_**1124**_in. It should be noted that each of input port mapping comments **1161** and **1162** includes a pre-pended

non-conventional comment identifier,--!!, that is utilized by the HDL compiler (such as compiler **462** in FIG. **4D**) to maintain the port mapping comments separate from the design.

To facilitate connection of a simulation event to an instrumentation entity input port, input port mapping comments **1161** and **1162** consist of two distinct parts: an instance identifier and an event identifier. The instance identifier is a string consisting of instance names (in descending hierarchical order) of all design entities between and including the design entity containing the instrumentation entity of the cross-hierarchical event being defined (i.e., the highest level design entity for the cross-hierarchical event), and the design entity in which the event that is utilized in generating the cross-hierarchical event. If the design entity containing the hierarchical event is the same as the design entity containing the event to be connected to, the instance identifier is a null string. A pair of instance identifiers **1163** and **1164** within input port mapping comments **1161** and **1162**, respectively, specify that events **1124** and **1108** originate from signals within design entity **1120** and **1104**, respectively.

Input port mapping comments **1161** and **1162** further include event identifiers **1165** and **1166** that identify input simulation events in terms of local instrumentation entities **1106** and **1122**, respectively. In accordance with the embodiment depicted in FIG. **11B**, each event identifier consists of a string beginning with an open bracket ("[") character and ending with a closed bracket ("]") character. Between these brackets, three sub-strings, delineated by period (".") characters, comprise a data structure utilized to identify a specific event from which the cross-hierarchical event is defined. The first sub-string within an event identifier is the instance name of the instrumentation entity containing the event. The second sub-string is a string specifying the type of the event ("count", "fail", "harvest", or "phase"). Finally, the third sub-string is the event name of the given event as specified in the declaration comment for the event. Each event identifier string uniquely identifies a single event within a given design entity. As depicted in FIG. **11B**, event identifier strings **1165** and **1166** identify events **1108** and **1124** respectively.

In accordance with an alternate embodiment of the present invention, the event identifier naming structure is modified slightly for events that are labeled in accordance with FIG. **10D** (event names that do not include the instrumentation entity name). When an instrumentation identifier is absent from the extended event identifier, the event identifier string with an input port mapping comment consists of two sub-strings: a string denoting the type of event to connect to; and a string providing the name of the event separated by a period (".") character. The instrumentation entity name is not required in this case since all events of a given type associated with a given design entity will have unique names. The model build tools of the present invention will automatically search all instrumentation entities associated with the design entity called out by the instance identifier to determine which instrumentation entity generates an event having the name and type provided in the event identifier string.

Referring to FIG. **11C**, there is illustrated a set of data structures for performing hierarchical processing of simulation model events in accordance with a second embodiment of the present invention. In the depicted embodiment, a pair of input port mapping comments **1171** and **1172** employs syntax compatible with the event naming data structure depicted in FIG. **10D**.

Input port mapping comment **1171** connects event **1108** to input port event_**1108**_in on instrumentation entity **1150**. Likewise, input port mapping comment **1172** connects event

1124 to input port event_**1124**_in on instrumentation entity **1150**. By utilizing the augmented syntax of FIG. **11B** or FIG. **11C**, it is possible to create hierarchical events by connecting the inputs of instrumentation entities to events within the simulation model.

The above described system and method provides for practical instrumentation of simulation models and allows for efficient implementation of instrumentation logic through embedded instrumentation entities. Embedded instrumentation entities, as described hereinabove, are however necessarily limited to task-specific implementations. As described with reference to FIGS. **12A** and **12B**, the present invention further provides for a more flexible implementation of instrumentation logic in a more unstructured manner.

It is often necessary to tailor instrumentation logic to address unique problems and circumstances. Instrumentation logic of a specific and yet non-predefined nature that is designed in accordance with the techniques disclosed herein with reference to FIGS. **12A** and **12B** is referred herein as "random instrumentation logic." A data construct consisting of general logic primitives (Boolean operators, storage elements, etc.) and an interconnection method for these primitives is utilized for implementing such random instrumentation logic.

For instrumenting a simulation model as described heretofore, an HDL such as VHDL or Verilog is utilized as a platform from which instrumentation logic is generated. Appropriate instrumentation entity descriptor comments within design entity source code files couple the resultant instrumentation entities to designated target design entities within a simulation model.

In addition to entity descriptor comments within a design entity source code file, the foregoing instrumentation technique requires a separate HDL file in which the instrumentation entity is described. As explained with reference to FIGS. **12A** and **12B**, the present invention provides a method, system, and data structure for instrumenting design entities within a simulation model while avoiding the design process overhead required for creating a separate instrumentation entity HDL file.

In accordance with the teachings of the present invention, random instrumentation logic is directly deployed within target design entities in terms of individualized and customizable instrumentation descriptor comments. Such instrumentation descriptor comments are encoded within the target design entity HDL source code file and provide a means for the describing random instrumentation logic, events, and interconnections between the created instrumentation logic and the target design entity. The random instrumentation logic is inserted into the simulation model in a manner similar to the techniques used for embedded instrumentation entities to produce an instrumentation entity without the need for the creation of an explicit HDL instrumentation entity file.

With reference to FIG. **12A**, there is illustrated a representative target design entity **1200** wherein random instrumentation logic is implemented in accordance with a preferred embodiment of the present invention. Instantiated within target design entity **1200** is a design entity **1201**. As further depicted in FIG. **12A**, an instrumentation entity **1202** is instantiated within design entity **1201**. Instrumentation entity **1202** is designed in accordance with the principles set forth hereinabove to generate a count event **1203** having an event name "count1." Target design entity **1200** further includes an instrumentation entity **1208** that is generated utilizing random instrumentation logic. As depicted in FIG. **12A**, instrumentation entity **1208** receives as inputs signals P, R, A, B, C and D along with count event **1203**.

Instrumentation entity **1208** is constructed by a set of unconventional comments lines within the source code file for target design entity **1200**. These comments may be incorporated at any point within the logic description section of the HDL source code file. HDL compiler **462** (FIG. **4**B) recognizes the unconventional comments in addition to any comments utilized to instantiate embedded instrumentation entities within design entity **1200**. During the post-compilation/model build phase, instrumentation load tool **464** processes these comments in a manner similar to that utilized for embedded instrumentation entities (described with reference to FIGS. **10**A-**10**D) to generate instrumentation entity **1208**.

A variety of possible syntaxes can be utilized to formulate the unconventional HDL comments required for generating random instrumentation logic within the source code file of a target design entity. As depicted in FIG. **12**B, much of the syntax of these comments employs syntax similar to the concurrent subset of the VHDL language with the addition of syntactic and semantic enhancements that provide a means of connection between an instrumentation entity and its target design entity. In addition, minor syntactic and semantic enhancements are provided to declare events and intermediate signals.

With reference now to FIG. **12**B, there is illustrated an exemplary HDL source code file **1220** that describes design entity **1200** of FIG. **12**A. Within HDL source code file **1220**, an entity instantiation **1221** produces design entity **1201**, and assignment statements **1222** are utilized to generate signals A, B, C and D. A set of unconventional comments **1223** within HDL source code file **1220** is utilized to produce instrumentation entity **1208**. Comments **1223** are formulated as left-hand side (l.h.s.)/right-hand side (r.h.s.) assignment statements of the form:

$$\{\text{l.h.s.}\} <= \{\text{r.h.s}\};$$

where {l.h.s.}, referred to hereinafter as lhs, is the assignment statement target and, {r.h.s}, referred to hereinafter as rhs is an expression denoting the logical value to be assigned to the statement lhs. A number of rules delineate the possible expressions for lhs and rhs in any legal statement in the instrumentation comments.

As employed within the instrumentation data structure of the present invention, an lhs statement may be either an event declaration or the name of a signal that is instantiated within an instrumentation entity. An event declaration is an expression within bracket characters ("[A, A]") that generates a new event. Within comments **1223**, a statement **1230** produces a count event **1240** from instrumentation entity **1208** (FIG. **12**A) having eventname "countname0".

Within an lhs event declaration, a first field designates the event type (count, fail, harvest, phase, etc.) and is followed by such other fields as are necessary to declare the event. As illustrated in lines **1230**, **1234**, and **1236**, such event declaration fields follow the same format as the event declaration fields depicted in FIG. **4**C.

Comments **1223** further include a line **1232** having an lhs that declares a signal Q within instrumentation entity **1208**. To prevent ambiguity, any signal declared in this manner may not have a name corresponding to the name of any signal present on the top level of target design entity **1200**. Conformance to this requirement is verified by instrumentation load tool **464** (FIG. **4**D) during processing. Signals declared by an lhs expression may be incorporated within an rhs expression as shown in lines **1232** and **1234**.

An rhs consists of logical connectivity expressions and/or functions that combine various signals. Signals within these

connectivity expressions may originate from a number of possible sources including: signals declared on the lhs of a statement in the instrumentation comments; signals within the target design entity; or signals designating other events within the target design entity.

The absence of period (".") or bracket ("[", "]") characters within a signal value description in the rhs of a statement, designates the object signal as corresponding to either a signal within the top hierarchical level of the target design entity or to a signal declared on the lhs of a statement within the instrumentation language. Signals are named in a mutually exclusive manner by the rules governing creation of signals on the lhs of a statement in the instrumentation comments, thereby preventing any ambiguity in the determining the source of the given signal.

Signals in rhs connectivity expressions can also be connections to signals within entities instantiated within the target design entity. In such a circumstance, the instance names of the entity or entities in the hierarchy enclosing the desired signal are placed before the signal name in hierarchy order, delineated by period (".") characters. For example, the signal in statement **1230** ("Y.P") represents signal **1204** within design entity **1201**. Signals at any level of the target design hierarchy are thus accessible to instrumentation logic generated by the instrumentation language comments.

Signals within the instrumentation comment expressions can also designate other events within the target entity. Event identifiers as described hereinbefore for hierarchical events are used to denote such "event" signals. For example, statement **1232** performs a logical AND of instrumentation event **1203** and signal A. The event identifier "Y.[B1.count.count1]" connects instrumentation entity **1208** to instrumentation event **1203**. This notation permits instrumentation events at any level of design hierarchy within target design entity **1200** to be directly accessed.

As further depicted in FIG. **12**B, statement **1232** produces intermediate signal Q within instrumentation entity **1208**. This is an example of an instrumentation comment statement declaring a new intermediate signal. These signals can be used in other statements to construct random instrumentation logic of any desired depth or complexity.

Statement **1234** utilizes intermediate signal Q along with signal **1206** to produce fail event **1241**. The syntax for fail event declaration includes a field denoting the type of event ("fail"), a field giving the event name for the fail event ("failname0"), and a final field denoting the message to associate with the fail. Statement **1236** produces harvest event **1242** by including within random instrumentation logic **1208** an AND gate that performs the specified logical operation B AND C. Finally, statement **1238** produces phase event signal **1250**. In the case of phase event declaration, the lhs assignment expression target is of the form: [<eventtype>, <eventname>, <phasename>], where <eventtype> is "phase", <eventname> is the name of the phase event, and <phasename> is the aggregate phase name. As described above, the external phase event signal **1250** generated by instrumentation entity **1208** will be combined with other phase events sharing the same aggregate phasename by an instance of combining logic **427**. The instance of combining logic **427** implements the combining function specified for the aggregate phasename in phase file **492** (or predefined for the phasename) in order to generate the aggregate phase event signal **429** for the aggregate phasename (in this case end-of-testcase).

In general, the rhs expression of any statement in the instrumentation data structure of the present invention can access any signal or instrumentation event signal within the target design entity utilizing these syntactic mechanisms. These

signals can be combined to form new events or intermediate signals that can themselves be further combined to form instrumentation logic of any desired depth or complexity.

Instrumentation comments can be placed anywhere within the logic description section of the target entity source code file. All instrumentation comments within a file are considered as a whole and produce a single instrumentation entity within the target design entity.

Simulation of a given model is typically controlled by a program, hereinafter referred to as RTX (Run Time eXecutive), that is written in a high-level language such as C or C++. To facilitate RTX control and execution of a simulation run, simulators typically provide a number of application program interface (API) functions that may be called by the RTX. Such API functions employ routines, protocols, and tools that allow for polling of signals within the simulation model, alteration of signals within a simulation model, cycling a simulation model, etc.

The RTX is often required to monitor occurrences of significant events during model simulation. Such events typically consist of a signal or a set of signals that assume a prescribed sequence of values and will be referred to hereinafter as "model events." To monitor model signals, an RTX typically calls a specialized API function, hereinafter referred to as GETFAC, which allows for polling of signal values during model simulation.

Typically, an RTX must monitor a large number of signals, potentially on a cycle-by-cycle basis, within a simulation model and must subsequently process the signal values in order to detect the occurrence of model events. This approach to model event monitoring places a burden on the verification engineer in terms of re-writing RTX and communicating with design engineers when the signals or sequence of signal values that constitute a model event change.

To provide an efficient means for monitoring model events, so-called "detection events" are generated and are accessible by the RTX. Such detection events are generated by instrumentation entities. Detection events are implemented as output ports on an instrumentation entity. Furthermore, an enhanced API function is provided for directly accessing detection events within a given simulation model.

For each detection event, a first, potentially multi-bit, output is utilized as the value of a model event. An optional second single bit signal is utilized to indicate when the detection event occurs during model simulation. By their nature, certain model events occur at each and every cycle and therefore do not require a qualifying signal to track their occurrences.

With reference to FIG. **13A**, there is illustrated a block diagram depicting data content within main memory **44** (FIG. **2**) during a simulation run of a simulation model **1400**. Main memory **44** includes the elements necessary for monitoring an exemplary model event including a software simulator **1410** that simulates simulation model **1400** under the control of an RTX **1405**.

RTX **1405** delivers a set of API calls **1430** to API function GETFAC within simulator **1410** to obtain the values of signals A, B, C, and D within model **1400**. Further processing of these signal events is performed utilizing an RTX code **1450** culminating in the assignment of the model event value to variable event_x at RTX code line **1455**.

Referring to FIG. **13B**, there is illustrated a block diagram depicting contents of main memory **44** during a simulation run in accordance with a preferred embodiment of the present invention. In the depicted embodiment, an instrumentation entity **1460** is instantiated within simulation model **1400** using techniques described above. Instrumentation entity

1460 directly monitors signals A, B, C, and D by means of a set of signal connections **1462**. Signal connections **1462** provide a more efficient means to monitor signals than GETFAC API function calls.

Within instrumentation entity **1460**, instrumentation logic **1464** substantially recreates the function of RTX code **1450** of FIG. **13A** and produces signals **1466** and **1468**, which denote the value of a model event (e.g., a fail, count, harvest or phase event) and when the model event occurs, respectively.

Each detection event within a given simulation model is assigned a unique name in a manner described below. During model build, instrumentation load tool **464** (FIG. **4D**) generates a data structure in the form of a table within the simulation model that uniquely names all the detection events within a given simulation model and the corresponding instrumentation entity output signals. This table will be referred to hereinafter as the detection event translation table.

An API function GETEVENT( ) is provided within software simulator **1410** for accessing model detection events. API function GETEVENT references a detection event translation table **1470** to locate signals **1466** and **1468** in response to a call **1472** by RTX to obtain the value of model event event_x. RTX **1405** obtains the value of model event event_x without delivering a number of GETFAC API calls, and furthermore, without the need to process the signal values associated with the model event. The RTX code is thus insulated from potential changes to the signals and signal sequence values defining model event event_x. Any changes to the detailed definition of model event event_x are reflected within instrumentation entity **1460** and no changes to the RTX are necessary.

It will further be appreciated that one or more variants of the API function GETEVENT( ) may be defined to support different parameter sets for different types of events. For example, a first API function GETPHASE (phasename, instance) can be defined to obtain the logic value of the specified phase event, as optionally further qualified by an instance parameter identifying a particular target design entity instance within the simulation model. A second API function GET_AGG_PHASE (phasename) can further be defined to obtain the logic value of the aggregate phase event signal **429** for the specified aggregate phasename.

With reference to FIG. **13C**, there is illustrated an exemplary HDL source code file **1480** that describes instrumentation entity **1460** in accordance with a preferred embodiment of the present invention. As shown in FIG. **13C**, exemplary file **1480** consists of a number of entity descriptor comments **1491** and an architecture section **1492** comprising instrumentation logic **1464**.

Within HDL file **1480**, a set of input port map comments **1493** serve to generate connections **1462** of FIG. **13B**. An additional comment section, detection declarations **1494** is incorporated within the entity descriptor comment syntax that allows for declaring detection events. A detection declaration comment **1495** serves to generate and uniquely name detection event event_x. Moreover, detection declaration comment **1495** associates signals **1466** and **1468** of FIG. **13B** with event_x. Detection event declarations, such as detection event declaration **1495** are of the form:

```
--!!<name>: event_value_port [ctrl_signal];
```

where name is a name associated with the specific detection event (event_x in FIG. **13C**), event_value_port is the output

port providing the value for the detection event, and ctrl_signal is an optional single bit output port that flags an occurrence of the model event.

Each detection event is uniquely named in accordance with the name field within the output declaration comment in a manner analogous to that described earlier for count events. Such detection event names, together with the corresponding instrumentation entity output ports, are inserted into the detection event translation table data structure that is placed within the model by instrumentation load tool 464. API function GETEVENT receives the extended event identifier associated with a given event as an input and returns the model event value and, if applicable, an indication of whether the event occurred in the current cycle.

While FIG. 13C illustrates only those constructs necessary for implementing a detection event, the principles set forth herein place no limitation on the generation of count, fail, harvest or phase events or signal overrides within the same instrumentation entity as a detection event. Moreover, multiple detection events may be incorporated within the same instrumentation entity.

Within the spirit and scope of the present invention, detection events may be created within random instrumentation comments in a manner largely similar to that described with reference to signal overrides. Detection events can also be combined, in a manner similar to that shown earlier, as part of a hierarchical event.

Finally, it should be noted that the present invention may be practiced in conjunction with a hardware simulator. As for software simulators, hardware simulators are controlled by an RTX program. To adapt the principles set forth herein to a hardware simulator environment, the hardware simulator provides a GETEVENT API function and accepts models containing a detection event translation table.

By utilizing random instrumentation comments, a design engineer can efficiently create representations of model events accessible to RTX. Such representations need not change even if the detailed definition of the model event changes. Such stability reduces the complexity and burden of maintaining RTX and lowers the amount of communication required between design and verification engineers.

In order to provide for the control and monitoring of instrumentation events within simulation models executing on a batch simulation farm, one or more general-purpose computers, hereinafter referred to as "instrumentation servers", are added to batch simulation farms. An instrumentation server acts as a centralized repository for information used to control instrumentation events and for data gathered from instrumentation events during simulation runs. The exact nature and function of the control information and of the gathered data varies with the type of event (i.e., fail events vs. count events), as will be described below.

In order to allow for effective management of instrumentation events, a set of "eventlist" files (described with reference to FIGS. 10A-10D) contain information about the exact number and content of the instrumentation events in a given model. The eventlist files are created at model build time by instrumentation load tool 464. These files, one per class of events (fail, count, harvest, phase etc.), list the particular events in the given simulation model. Each simulation model has a unique set of corresponding eventlist files that are created at model build time.

When instrumentation events are created at model build time, they are constructed in a specific order, and a unique index is assigned within the eventlist file to each instrumentation event for a given event class. Accesses to instrumentation events by API routines make use of these index values.

Furthermore, when an API routine communicates aggregate data with respect to all events within a given event class to the instrumentation server, this aggregate data is sequentially ordered according to these index values.

Each eventlist file contains a list of the instrumentation events for a given event class within the model. These events are named in accordance with the naming convention data structures described above in conjunction with FIGS. 10A-10C, which provides unique names for each of the instrumentation events. Referring back to FIG. 10A in conjunction with FIG. 14, there is shown an eventlist file 1660 for the count events of simulation model 1000 of FIG. 10A.

Eventlist file 1660 contains multiple count event class entries 1663. Each of count event class entries 1663 includes a unique, sequential index value 1661, and an extended event identifier 1662. Each of indices 1661 corresponds to the index of a particular event (in this case count event COUNT1) assigned at model build time. Extended event identifiers 1662 provide an event name associated with each individual event index. Eventlist file 1660 thus provides a mapping between the instrumentation event names and the instrumentation event indices as well as providing an ordering convention for aggregate data for a class of instrumentation events. Eventlist files, such as eventlist file 1660, are used by the instrumentation server to aid in the control and monitoring of instrumentation events in simulation models.

As has been described, the present invention provides a method, system and program product for simulation processing. According to one embodiment of the invention, an instrumented simulation executable model of a design is built by compiling one or more hardware description language (HDL) files specifying one or more design entities within the design and one or more instrumentation entities and instantiating instances of the one or more instrumentation entities within instances of the one or more design entities. Operation of the design is then simulated utilizing the instrumented simulation executable model. In the simulation, each of multiple instantiations of the one or more instrumentation entities generates a respective external phase signal representing an occurrence of a particular phase of operation, and instrumentation combining logic generates from external phase signals of the multiple instantiations of the one or more instrumentation entities an aggregate phase signal representing an occurrence of the particular phase of operation, such as an end-of-testcase.

While the invention has been particularly shown as described with reference to a preferred embodiment, it will be understood by those skilled in the art that various changes in form and detail may be made therein without departing from the spirit and scope of the invention. For example, one of the embodiments of the invention can be implemented as sets of program code resident in random access memory 28 of one or more computer systems configured generally as described in FIG. 1 and FIG. 2. Until required by computer system 10, the set of program code may be stored in another computer readable storage device, such as disk drive 33 or in a removable storage device such as an optical disk for eventual use in a CD-ROM drive or a floppy disk for eventual use in a floppy disk drive. The set of program code embodied in a computer usable medium may be referred to as a computer program product. Further, the set of program code can be stored in the memory of another computer and transmitted over a local area network or a wide area network, such as the Internet, when desired by the user.

What is claimed is:

1. A method of simulation processing in a data processing system, said method comprising:

building an instrumented simulation executable model of a design, said building including compiling one or more hardware description language (HDL) files specifying one or more design entities within the design and one or more instrumentation entities and instantiating instances of the one or more instrumentation entities within instances of the one or more design entities, wherein said step of instantiating comprises instantiating each of the instances within a particular design entity indicated by the one or more HDL files, and wherein said instrumented simulation executable model includes a plurality of instances of said one or more design entities;

simulating operation of the design utilizing the instrumented simulation executable model, wherein said simulating includes:

each of multiple instantiations of the one or more instrumentation entities generating from one or more signals in at least one instance of the one or more design entities a respective external phase signal representing an occurrence of a particular phase of operation, wherein said particular phase is an end-of-testcase phase indicating that an end of a simulation testcase has been reached;

identifying each of a plurality of phases of operation of said instrumented simulation executable model with a respective phasename;

restricting application of at least one phasename to a region of the instrumented simulation executable model defined by one of said plurality of instances of said one or more design entities;

instrumentation combining logic generating from external phase signals of the multiple instantiations of the one or more instrumentation entities an aggregate phase signal representing an occurrence of the particular phase; and

the data processing system automatically halting simulation of the design in response to the aggregate phase signal indicating an occurrence of the end-of-testcase phase.

2. A program product, comprising:

a computer-readable storage medium; and

model build program code within the computer-readable storage medium that, when processed by a data processing system, builds an instrumented simulation executable model of a design by performing a model build process including:

compiling one or more hardware description language (HDL) files specifying one or more design entities within a design and one or more instrumentation entities for instrumenting the design entities for simulation;

generating instrumentation combining logic that aggregates external phase signals of instances of the one or more instrumentation entities; and

integrating instances of the one or more instrumentation entities and the instrumentation combining logic into instances of the one or more design entities to obtain the instrumented simulation executable model of the design, wherein in the instrumented simulation executable model, multiple instantiations of the one or more instrumentation entities are each coupled to receive as inputs one or more signals in instances of

the one or more design entities and to produce as an output a respective external phase signal representing an occurrence of a particular phase of operation, and wherein said instrumentation combining logic is coupled to receive as inputs the external phase signals and produce as an output an aggregate phase signal representing an occurrence of the particular phase;

wherein:

said model build program code instantiates the instantiations of the one or more instrumentation entities in instances of one or more design entities indicated by the one or more HDL files;

said particular phase is an end-of-testcase phase indicating that an end of a simulation testcase has been reached.

said instrumented simulation executable model includes a plurality of instances of said one or more design entities; and

said model build program code identifies each of a plurality of phases of operation of said instrumented simulation executable model with a respective phasename and restricts application of at least one phasename to a region of the instrumented simulation executable model defined by one of said plurality of instances of said one or more design entities.

3. A program product, comprising:

a computer-readable storage medium; and

an instrumented simulation executable model of a design within the computer-readable storage medium, said instrumented simulation executable model including:

instances of one or more design entities defining the design,

instantiations of one or more instrumentation entities that detect occurrences of specified event within the instances of the one or more design entities, and

instrumentation combining logic, wherein multiple instantiations of the one or more instrumentation entities are each coupled to receive as inputs one or more signals in instances of the one or more design entities and to produce as an output a respective external phase signal representing an occurrence of a particular phase of operation, and wherein said instrumentation combining logic is coupled to receive as inputs the external phase signals and produce as an output an aggregate phase signal representing an occurrence of the particular phase,

wherein:

when simulated by a software simulator, said instrumented simulation executable model records the occurrence of the particular phase;

said particular phase is an end-of-testcase phase indicating that an end of a simulation testcase has been reached;

the instrumented simulation executable model automatically signals to halt simulation of the design in response to the aggregate phase signal indicating an occurrence of the end-of-testcase phase; and

said instrumented simulation executable model reports said occurrence of the particular phase in response to a call querying for information regarding the particular phase.

* * * * *